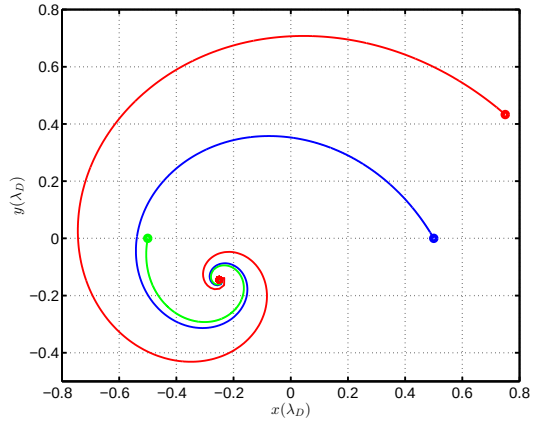
UiO : Department of Physics
University of Oslo

Two dimensional vortex structures in magnetized plasmas

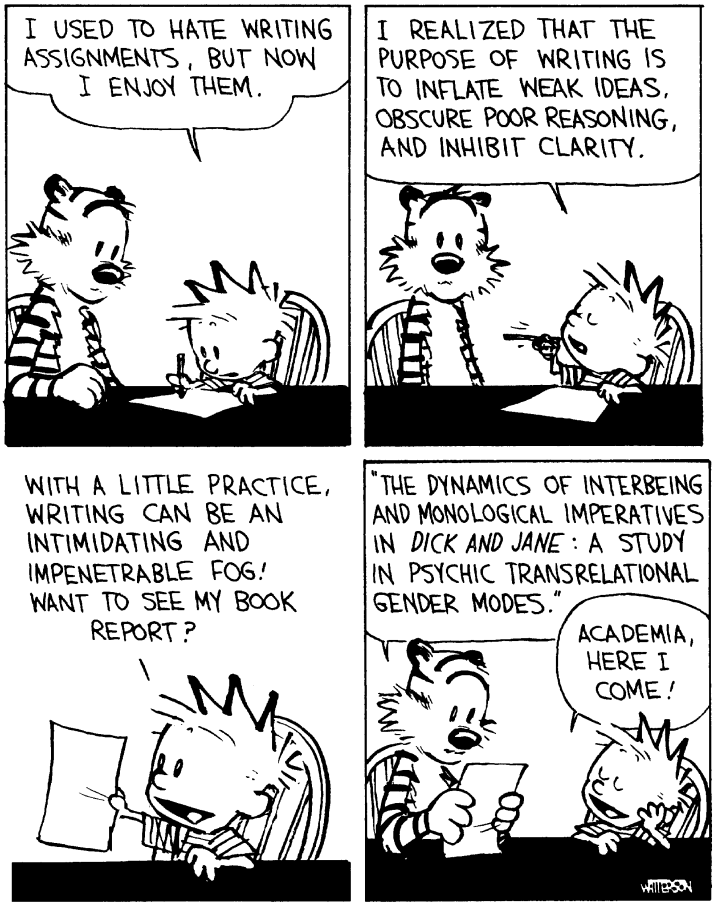
Master Thesis

Hans Brenna

February 2013



The figure on the front page shows the collapsing trajectories of three vortices with strengths $\gamma_1 = 2$, $\gamma_2 = 2$, $\gamma_3 = -1$ *at the positions* $(1/2, 0)$, $(-1/2, 0)$ *and* $\sqrt{3}/2(\cos(\pi/6), \sin(\pi/6))$


I USED TO HATE WRITING ASSIGNMENTS, BUT NOW I ENJOY THEM.
I REALIZED THAT THE PURPOSE OF WRITING IS TO INFLATE WEAK IDEAS, OBSCURE POOR REASONING, AND INHIBIT CLARITY.
WITH A LITTLE PRACTICE, WRITING CAN BE AN INTIMIDATING AND IMPENETRABLE FOG! WANT TO SEE MY BOOK REPORT?
"THE DYNAMICS OF INTERBEING AND MONOLOGICAL IMPERATIVES IN *DICK AND JANE*: A STUDY IN PSYCHIC TRANSRELATIONAL GENDER MODES."
ACADEMIA, HERE I COME!
WATTERSON

Abstract

This thesis presents a numerical study of vortex structures in magnetized plasmas. In a first approximation, such vortices can be understood as a collection of magnetic field aligned charged filaments. Their charge distribution gives rise to slowly varying $\mathbf{E} \times \mathbf{B}/B^2$-drifts of the ambient plasma and the vortices embedded there. A mathematical model has been derived, studied analytically for low-dimensional vortex systems and implemented as computer code. The code has been verified by recreation of some of the analytical results.

The main focus has been on the study of macroscopic structures created by superimposing many discrete point vortex systems and on the study of homogeneous and isotropic vortex systems approximated by periodic boundary conditions. The dynamics of the structures show a wealth of phenomena for relatively simple model, including long lived coherent formations and the evolution of stable tripolar macroscopic vortex systems from the collision of two vortex pairs. Homogeneous and isotropic vortex systems display the basic properties of turbulent diffusion and transport, i.e. finite correlation time, continuous power spectra, etc. From these results we have calculated effective diffusion coefficients for a range of vortex strengths and we have found phenomenological relations between the Eulerian and Lagrangian integral time scales and mean square velocities.

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Hans Pécseli for giving me this opportunity, for all your support and help during this project, and for reading my thesis innumerable times. It has been a great experience working with you!

I would also like to thank Vegard Lundby Rekaa for all his help with the writing of my code. Thank you for taking your time to explain, and helping me debug when I thought I had exhausted all options. Also, thank you for the proofreading in the final stages.

And Prof. em. Jan Trulsen for his invaluable advise with the numerics and for helping me understand things I had never even thought of.

Thanks to Bjørn Lybekk for making several illustrations used in this thesis.

Thanks to Anne Bregsaker, Elling Hauge-Iversen, Christoffer Stausland and all the other students and staff at the Plasma and space physics for interesting discussions and good times, during lunch and at times when one should work on one's thesis.

And to I-M for making me think of other things occasionally.

Finally I would like to thank everyone who has made me explain what I've been doing this past year. Without failing to answer that question repeatedly, I would probably never have understood it myself.

Contents

Chapter 1

Introduction

1.1 Plasma

A plasma is a state of matter where the atoms or molecules have been partly or fully ionized, so that a portion of the ions and electrons can move about freely. Plasma is in many ways similar to a gas, i.e. it will have no determined volume or shape unless placed in a container. Under the influence of electric fields, the high conductivity of plasma leads flows of charged particles generating currents and magnetic fields. Under the influence of magnetic fields it can form structures such as filaments, rays and double layers.

In contrast with neutral gases, where the interactions between particles are short range, long range interactions and forces due to electric and magnetic fields are important in describing the properties and dynamics of plasmas. This makes plasmas so different from gases that plasma is often called the fourth state of matter.

Plasma is the most common state of ordinary matter in the universe, over 90% of it is in a plasma state, mainly in the intergalactic medium. Most of the visible matter is ionized as well, in stars. On Earth, plasma is much less common and is usually associated with human activity, though lightning and other processes in thunder storms are known to produce plasma. Plasma is common, though in earth's upper atmosphere, as well as near and outer space. Because of the rarity of plasma phenomena on earth, it was not discovered until the 1870s by Sir William Crookes during his experiments on Crookes tubes (Crookes). Crookes used the term *radiant matter* for what he observed; the term plasma was first used in 1928 by Irving Langmuir (Langmuir, 1928).

Figure 1.1 illustrated the range of temperatures and densities at which plasmas can exist. Except for the plasmas in stars, and especially stellar

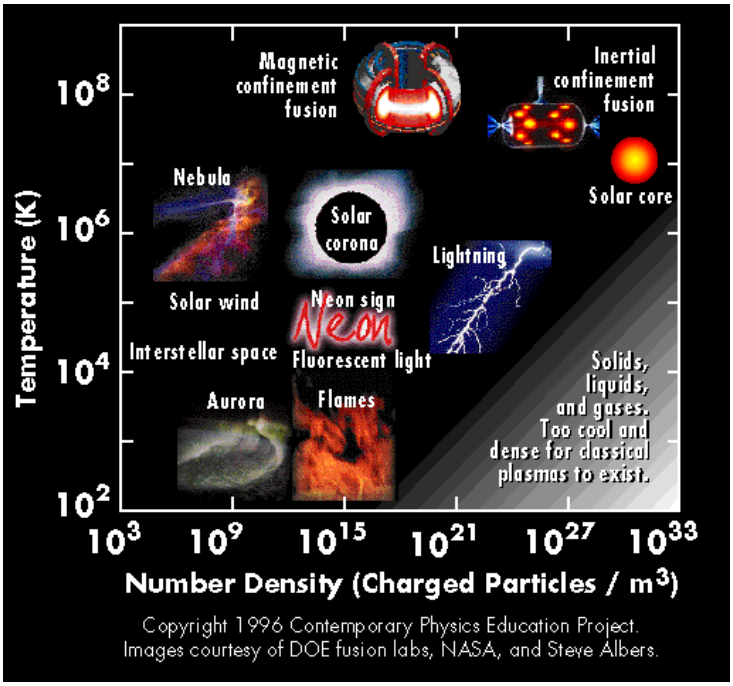


Figure 1.1: *Plasma can exist at a large range of temperatures and densities.*

cores, natural space plasmas tend to be of low density and often hot.

Since plasmas are so common i space, studies of plasma phenomena can be motivated by the need to have equipment working in environments dominated by plasma for extended periods. Studies of the interaction between a plasma and an embedded object is an active area of research.

Artificial plasmas have a wide range of industrial applications, while the most notable application of plasma physics is, perhaps, the unsolved problem of sustained thermonuclear fusion for power generation.

1.2 Turbulence

Turbulence is one of the great unsolved problems in modern physics. Turbulent phenomena are characterized by chaotic dynamics and rapid transfer of energy between different length scales, and are in general, not solvable analytically. Numerical simulations is the main theoretical tool for the study of physically realistic turbulence, but numerical simulations of fully developed turbulence in three spatial dimensions poses a formidable problem for modern computers. The range, or complexity, of turbulent phenomena a numerical model is able to reproduce is often measured by the Reynolds number of the

initial flow. The range, or complexity, of turbulent phenomena a numerical model is able to reproduce is often measured by the Reynolds number of the initial flow. Some of the largest present day simulations thus assume Reynolds numbers ~ 500 or even less; compared to ~ 5000 (Frisch, 1996) that can be quite easily obtained in turbulent pipe flows. It can have great value to find simpler, yet physically realistic and realisable models, that require reduced computer resources. Such models can then be used as a "test-bed" for ideas of general interest, such as the Eulerian-Lagrangian transformation of time scales or correlation functions, detailed investigations of parameter variations of turbulent diffusion, etc. Simulations in two dimensions, as addressed in the present study, offer such a possibility.

1.3 Motivation

In this thesis , we will first discuss general plasma phenomena (scales, single particle motions, kinetic and fluid models) in Chapter 2. We will then introduce flute modes and turbulent diffusion as the theoretical framework needed to analyse the results we present later. Next we present the numerical methods developed for this thesis in Chapter 5 and the data analysis methods in Chapter 6. Last we present the results in Chapter 7 and some concluding remarks in Chapter 8.

The main goals of this thesis are:

- The derivation and description of a simple, yet realistic, model for the low frequency dynamics of homogeneously magnetized plasmas; formulated in terms of interacting line vortices.
- The implementation of this model in a computer program.
- The utilisation of the program to run simulations of physically relevant systems.
- Demonstrate that vortex systems can develop characteristics similar to turbulent flows, i.e. correlation functions with finite memory (correlation times) and continuous power spectra.
- Use numerical results to illustrate basic results for particle transport due to random or turbulent motions.

Chapter 2

Basics of plasma physics

2.1 Single particle motion

A good starting point for our discussion of basic plasma physics is the dynamics and behaviour of single particles interacting with electric and magnetic fields. Many of the drifts and phenomena described here are important as bulk movements in plasmas consisting of many particles, especially if the plasma is dilute.

2.1.1 The E × B-drift

The force on a charged particle in an electric and a magnetic field is given by the Lorentz force

$$\mathbf{F} = q(\mathbf{E} + \mathbf{U} \times \mathbf{B}). \tag{2.1}$$

where $\mathbf{F}$ is the force on a particle with charge q due to the electric field $\mathbf{E}$ and the magnetic field $\mathbf{B}$ and $\mathbf{U}$ is the velocity of the particle.

Consider a single particle with mass m moving in a uniform and stationary magnetic field with velocity $\mathbf{U}$. Assuming $\mathbf{E} = 0$ the equation of motion for this particle is then

$$m\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{U}_\perp = q\mathbf{U}_\perp \times \mathbf{B}, \tag{2.2}$$

where $\mathbf{U}_\perp$ is the components of the velocity which are perpendicular to $\mathbf{B}$. Since the magnetic force on a particle is always perpendicular to the velocity, magnetic forces alone can neither add, nor transfer, energy to or from the particle, only change the direction of the velocity. The result is a gyrating motion of the particle in a circular orbit with radius

$$r_L = \frac{mU_\perp}{qB}, \tag{2.3}$$

where r_L is known as the Larmor radius with corresponding frequency

$$\Omega_c = \frac{qB}{m} \tag{2.4}$$

also referred to as the cyclotron frequency.

Consider the case where we have a constant electric field in the direction perpendicular to $\mathbf{B}$, the magnetic field still being constant in space and time. The equations of motion in the direction perpendicular to $\mathbf{B}$ becomes

$$m\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{U}_\perp = q(\mathbf{E}_\perp + \mathbf{U}_\perp \times \mathbf{B}) \tag{2.5}$$

We know that by a suitable change of reference the electric field can be made to vanish. We introduce a new velocity $\mathbf{U}_* = \mathbf{U}_\perp - \mathbf{E}_\perp \times \mathbf{B}/B^2$, where $\mathbf{U}_\perp$ is the "true" particle velocity. Substituting this new velocity into the equation of motion correspond to changing the frame of reference to the one in which the electric field is vanishing. The velocity $\mathbf{U}_*$ thus follows the equation

$$m\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{U}_*(t) = q\mathbf{U}_*(t) \times \mathbf{B}. \tag{2.6}$$

In this frame of reference we have a gyro-orbit solution for $\mathbf{U}_*(t)$, as in equation 2.2. The actual trajectory is obtained by changing reference system back to the laboratory frame by adding the velocity

$$\overline{\mathbf{U}}_{\mathbf{E}\times\mathbf{B}} = \frac{\mathbf{E} \times \mathbf{B}}{B^2} \tag{2.7}$$

generally called the $\mathbf{E}\times\mathbf{B}$-velocity. This will be an average velocity associated with the gyro-centre, in addition to the circular gyrating motion. The real trajectory will thus be a type of curve called a *cycloid*.

Physically, the origin of the $\mathbf{E}\times\mathbf{B}$-drift is the following: Since a stationary magnetic field can not change the energy of a charged particle, the changes in velocity in the particle orbits are solely caused by the electric field, which accelerates ions in the positive field direction and electrons in the negative field direction. The radius of curvature, caused by the magnetic field, is small when the velocity is small and large when the velocity is large. The average drift results from this and is illustrated in Figure 2.1(B).

A useful generalisation is made by introducing the instantaneous drift velocity $\mathbf{U}(t)$. We will in this thesis consider magnetic fields that are homogeneous and stationary, while electric fields can vary in time, i.e. the instantaneous $\mathbf{E} \times \mathbf{B}$-velocity is given by

$$\mathbf{U}(t) = \frac{\mathbf{E}(t) \times \mathbf{B}}{B^2}. \tag{2.8}$$

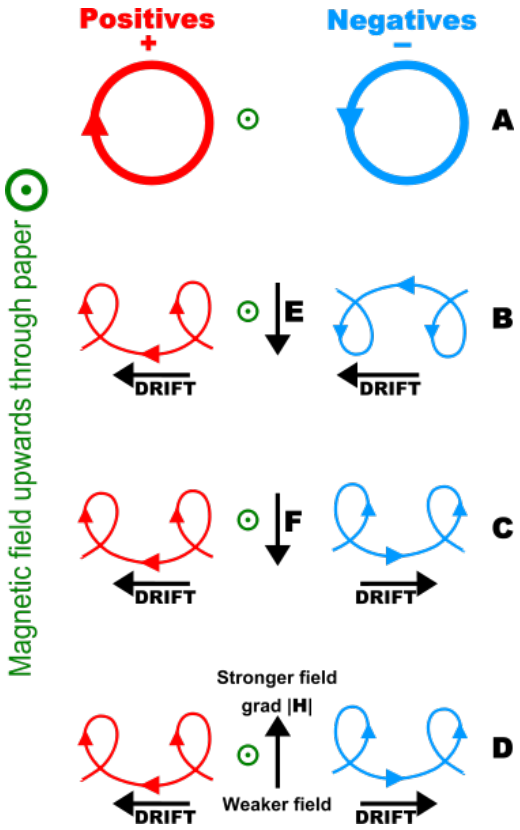


Figure 2.1: *Illustration of single particle drifts in a homogeneous magnetic field. (A) No disturbing force, (B) a homogeneous electric field, (C) an independent homogeneous force, e.g. a gravitational field, (D) an inhomogeneity in the magnetic field. Image created by Ian Tresman. Source:http://commons.wikimedia.org/wiki/File:Charged-particle-drifts.svg*

2.1.2 Other drifts

We are free to generalize the result from Section 2.1.1 to not just electrical forces, but for force in general co-interacting with the magnetic force on the particle. Substituting $\mathbf{E}$ with the electric force exerted on a particle of charge q i.e. $\mathbf{E} = \mathbf{F}/q$ we retrieve an expression for the drift caused by a general force $\mathbf{F}$ and magnetic field $\mathbf{B}$

$$\overline{\mathbf{U}} = \frac{\mathbf{F} \times \mathbf{B}}{qB^2}. \tag{2.9}$$

These drifts will in general result in charge separation due to the dependence on q.

Polarization drift

If the electric field is time varying, this gives rise to a drift called the polarization drift. Consider a particle moving with the instantaneous $\mathbf{E} \times \mathbf{B}/B^2$ velocity, in an electric field which varies in magnitude, but not in direction, $(dE/dt \neq 0)$. For an electric field which is slowly varying, the reference frame moving with the particle is not an inertial frame, this means that the particle will experience an acceleration which can be interpreted as an effective gravitational force $M\mathbf{g}$ originating from the variation in the $\mathbf{E} \times \mathbf{B}/B^2$. This effective, or virtual, gravity gives rise to an $M\mathbf{g} \times \mathbf{B}/(eB^2)$-velocity, varying with time, see (2.9). Since $\mathbf{B}$ is assumed constant, the force is $\mathbf{F} = M\mathbf{g} = -(M/B^2)(d\mathbf{E}/dt \times \mathbf{B})$. This can be inserted into (2.9) and gives

$$\mathbf{U}_{pol} = \frac{(-\frac{M}{B^2}\frac{\mathrm{d}\mathbf{E}}{\mathrm{d}t} \times \mathbf{B}) \times \mathbf{B}}{eB^2} = \frac{1}{\Omega_c}\frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathbf{E}}{B} \tag{2.10}$$

2.2 Basic plasma parameters

When considering plasmas significantly denser than those discussed in section 2.1, the collective behaviour of the plasma through the interaction of self-consisten electric and magnetic fields begin to dominate over single particle motions. There are four parameters that prove particularly useful for characterising a plasma. These are the thermal velocity, the plasma frequency, the Debye length and the plasma parameter. We will use these to construct characteristic spatial and temporal scales for our study. Therefore, brief definitions of these parameters are included here.

2.2.1 Thermal velocity

The thermal velocity is the typical velocity of the thermal motion of the particles in a fluid or gas of a given temperature. The definition used here is

$$u_{th,s} = \left(\frac{\kappa T_s}{m_s} \right)^{1/2}, \tag{2.11}$$

where κT_s is the temperature and m_s is the mass of particle species s. A numerical constant has been omitted from the definition for simplicity.

2.2.2 The plasma frequency

Consider a slab of plasma with the electron displaced slightly with respect to the ions. This small perturbation in the charge distribution sets up an electric field trying to restore the imbalance. Since the electrons are much lighter, and hence more mobile than the ions, the electrons start to oscillate around the equilibrium position with the characteristic frequency

$$\omega_{pe} = \left(\frac{e^2 n}{\varepsilon_0 m_e} \right)^{1/2} \tag{2.12}$$

where n is the number density and m_e is the electron mass. The corresponding plasma period is the $\tau_p = 2\pi/\omega_{pe}$.

2.2.3 The Debye length

The Debye length λ_D characterises a shielding distance. When a surplus particle with charge q is introduced into a plasma, the surrounding plasma reorganizes in an attempt to screen off the electric potential arising from the charge q. The result is that at larger distances from the perturbing charge, the perturbation is close to undetectable and only the collective behaviour of all the particles can be observed. At distance λ_D from the charge q, the electric potential of this charge is reduced by a factor e^{-1}: The charge is shielded by the surrounding plasma. The Debye length is defined as

$$\lambda_D = \left(\frac{\varepsilon_0 \kappa T}{e^2 n} \right)^{1/2}. \tag{2.13}$$

A plasma particle travelling with the thermal velocity will travel one Debye length in one plasma period.

2.2.4 The plasma parameter

From the Debye length and the plasma density the plasma parameter can be constructed as

$$N_p = n\lambda_D^3. \tag{2.14}$$

This dimensionless quantity is, apart from a factor of order unity, the average number of particles within a sphere with radius λ_D. For large N_p, a small perturbation in the plasma will not be noticed at large distances, since the surrounding plasma efficiently screens off the perturbation and the overall electric field is not noticeably influenced. If, on the other hand, N_p is low, any perturbation can have a significant effect on the surroundings. Plasmas with large N_p can for most purposes be considered to be collisionless and the dynamics are controlled by collective interactions.

Note that the plasma parameter actually decreases for increasing n assuming constant temperature, since $\lambda_D^3 \sim n^{-3/2}T^{3/2}$, we have $N_p \sim n^{-1/2}T^{3/2}$. Plasmas with large N_p thus have low density and high temperature, and are characterised as *hot* and *dilute*.

Note also that the plasma parameter as defined here only makes physical sense in three spatial dimensions. Since our main concern in this study is two dimensional plasmas, the plasma parameter has to be redefined as the number of particles in a rectangle with area λ_D^2.

2.2.5 Summary

Even though the parameters we have introduced here are constructed with an electron gas or electron dynamics as examples, it is straight forward to construct similar quantities for gases of different species and for mixed charged gases. In the latter example, different species are allowed to have different temperatures, densities, etc.

For the vortex structures introduced in the next chapter including the model and corresponding assumptions used, the thermal velocity and the plasma frequency are with limited physical significance. Instead the magnitude of the $\mathbf{E} \times \mathbf{B}/B^2$-velocity will serve as a characteristic velocity $U_0 = E_0/B_0$. The Debye length will still be used as a characteristic distance, and from combining the latter we can construct the characteristic time scale $t_0 = \lambda_D/U_0$ as the time a particle travelling with velocity U_0 takes to travel one Debye length.

2.3 Theoretical models

From the discussion so far, we can conclude that the motion of any charged particle in the presence of electric and magnetic fields will be governed by the Lorentz' force

$$\mathbf{F}(\mathbf{X}_j, t) = q_j(\mathbf{E}(\mathbf{X}_j, t) + \mathbf{U}_j(t) \times B(\mathbf{X}_j, t)), \tag{2.15}$$

where all fields are assumed to be functions of both position and time, e.g. $\mathbf{B} = \mathbf{B}(\mathbf{X}(t), t)$, and $\mathbf{X}_j = \mathbf{X}_j(t)$, $\mathbf{V}_j = \mathbf{V}_j(t)$, q_j are the position, velocity and charge of particle j. The magnetic and electric fields are given self consistently from Maxwell's equations

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0}, \quad \text{(Gauss' law for electric fields)} \tag{2.16}$$

$$\nabla \cdot \mathbf{B} = 0, \quad \text{(Gauss' law for magnetic fields)} \tag{2.17}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \quad \text{(Faraday's law)} \tag{2.18}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{j} + \mu_0 \varepsilon_0 \frac{\partial \mathbf{E}}{\partial t}, \quad \text{(The modified Ampere law)} \tag{2.19}$$

where ρ is the plasma charge density, $\mathbf{j}$ is the current density and ε_0, μ_0 is the vacuum permeability and susceptibility respectively. In all cases studied in this thesis the electric field will be assumed to be electrostatic, so $\mathbf{E} = -\nabla\phi$, and by substitution, Gauss' law for electric fields simplifies to Poisson's equation

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}. \tag{2.20}$$

2.3.1 Single particle description

Through integrating the equations of motion, a set of equations describing the trajectories of an ensemble $\mathcal{N}$ of particles can be retrieved, all of which are to be solved together with Maxwell's equations (2.16)-(2.19),

$$\frac{\mathrm{d}\mathbf{X}_j}{\mathrm{d}t} = \mathbf{U}_j \tag{2.21}$$

$$\frac{\mathrm{d}\mathbf{U}_j}{\mathrm{d}t} = \frac{\mathbf{F}_j}{m_j} = \frac{q_j}{m_j}(\mathbf{E}(\mathbf{X}_j, t) + \mathbf{U}_j(t) \times B(\mathbf{X}_j, t)) \tag{2.22}$$

$$\rho = \int q_j N(\mathbf{x}, \mathbf{u}, t) d\mathbf{v} \tag{2.23}$$

$$\mathbf{j} = \int q_j \mathbf{v} N(\mathbf{x}, \mathbf{u}, t) d\mathbf{v} \tag{2.24}$$

where $N(\mathbf{x}, \mathbf{v}, t) = \sum_j \delta(\mathbf{x} - \mathbf{X}_j(t))\delta(\mathbf{v} - \mathbf{V}_j(t))$ is the density of particles in phase space.

Attempting to solve these equations fully for the ensemble $\mathcal{N}$ involves solving equations (2.21)-(2.24) $\mathcal{N}^2$ times, a task which is in most cases almost impossible to perform numerically, let alone analytically. The challenge of solving such a problem is often referred to as the *N-body problem* and the method in general *molecular dynamics*. Here, it is worth mentioning the notation often used in numerical simulations, the order or complexity $\mathcal{O}(\mathcal{N}^2)$ for a given number of particles $\mathcal{N}$. Though computationally difficult to perform, the method has relevance to this thesis.

This single particle description, which explicitly keeps track of every single component particle in the plasma is, luckily, more detailed than usually needed, and there are two approximative descriptions of plasma dynamics that are widely used, and will be discussed in the remainder of this chapter..

2.3.2 Kinetic description

The first approximation is achieved by introducing the probability distribution function

$$f(\mathbf{x}, \mathbf{v}, t) = \langle N(\mathbf{x}, \mathbf{v}, t) \rangle \tag{2.25}$$

where $\langle ... \rangle$ represents the ensemble average over over all realisations of the plasma consistent with give constraints. By assuming that the exact solutions can be expanded in terms of a distribution and a correction to account for the two-particle interactions (collisions), we get the plasma kinetic equation. Assuming no collisions the kinetic equation simplifies to the Vlasov equation

$$\frac{\partial f(\mathbf{x}, \mathbf{v}, t)}{\partial t} + \mathbf{v} \cdot \nabla_x f(\mathbf{x}, \mathbf{v}, t) + \frac{q}{m} \{\mathbf{E}(\mathbf{x}, t) + \mathbf{v} \times \mathbf{B}(\mathbf{x}, t)\} = 0 \tag{2.26}$$

which describes the time evolution of a plasma. For the probability density functions $f(\mathbf{x}, \mathbf{v}, t)$ to represent a probabilistically acceptable system there are some requirements that need to be met:

- $f(\mathbf{x}, \mathbf{v}, t) \geq 0$ for all $\mathbf{x}, \mathbf{v}, t$.
- Integrating $f(\mathbf{x}, \mathbf{v}, t)$ over all physical space and velocity space gives the total number of particles.
- $f(\mathbf{x}, \mathbf{v}, t) \to 0$ for $v \to \pm\infty$.

The kinetic description has it's strengths in that it allows us to reduce the complexity of the problem without loosing all detailed information. It's strongest limitation is the assumption of no collisions, which in most space- and astrophysical problems, is after all a good approximation.

2.3.3 Fluid description

The fluid description of plasmas is even further removed from the single particle description, in that the fluid description only considers the bulk density and the bulk velocity, and as such describes the plasma as an electrically charged fluid. The quantities kept are the particle density, charge density, bulk velocity and charge density defined as:

$$n = n(\mathbf{x}, t) = \sum_s n_s = \sum_s \int f_s(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}, \tag{2.27}$$

$$\rho_e = \rho_e(\mathbf{x}, t) = \sum_s q_s n_s = \sum_s q_s \int f_s(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}, \tag{2.28}$$

$$n\mathbf{u} = n(\mathbf{x}, t)\mathbf{u}(\mathbf{x}, t) = \sum_s n_s \mathbf{u}_s = \sum_s \int \mathbf{v} f_s(\mathbf{x}, \mathbf{v}, t) d\mathbf{v}, \tag{2.29}$$

$$\mathbf{j} = \mathbf{j}(\mathbf{x}, t) = \sum_s q_s n_s \mathbf{u}_s \tag{2.30}$$

where s represents the particle species involved.

The fluid description can be derived from the kinetic description by multiplying the Vlasov equation with m, $m\mathbf{v}$ and $m\mathbf{vv}$ and then integrating over velocity space. Thus gives the fluid equations for conservation of mass, momentum and energy:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.31}$$

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = \rho_e (\mathbf{E} + \mathbf{u} \times \mathbf{B}) - \nabla p \tag{2.32}$$

$$\frac{\partial}{\partial t} \left(\frac{\rho u^2}{2} + \frac{3p}{2} \right) + \nabla \cdot \left(\frac{\rho u^2 \mathbf{u}}{2} + \frac{3p\mathbf{u}}{2} \right) = \mathbf{j} \cdot \mathbf{E} - \nabla \cdot (\rho \mathbf{u}). \tag{2.33}$$

Here $\rho = nm$ is mass density, p, $\rho u^2/2$ and $3p/2$ are the plasma pressure, kinetic energy and thermal energy, respectively, and ρ_e is the charge density. By combining equations (2.31)-(2.33) with Maxwell's equations and an equation of state, we have a closed set of equations which can be solved for a given plasma.

The most commonly used fluid plasma model, magnetohydrodynamics (MHD), treats the plasma as a single fluid, which makes charge separation impossible. There are different ways of omitting this limitation in fluid models, the simplest of which decscribe the plasma as consisting of multiple fluids, one for each particle specie. This allows for charge separation, but is generally more complicated.

Chapter 3

Flute modes

Flute modes are a limiting case of electrostatic plasma perturbations where all of (or almost all) spatial variations of the potential is in the direction perpendicular to the magnetic field $\mathbf{B}$. This limit can be derived form a more general three dimensional plasma model.

First we introduce the continuity equation

$$\frac{\partial n_e}{\partial t} + \nabla \cdot n_e \mathbf{U}_e = 0 \tag{3.1}$$

for the electrons, and

$$\frac{\partial n_i}{\partial t} + \nabla \cdot n_i \mathbf{U}_i = 0 \tag{3.2}$$

for the ions; and the momentum equations

$$m_e n_e \left(\frac{\partial \mathbf{U}_e}{\partial t} + \mathbf{U}_e \cdot \nabla \cdot \mathbf{U}_e \right) = -\nabla p - e n_e \left(\mathbf{E} + \mathbf{U}_e \times \mathbf{B} \right) \tag{3.3}$$

for the electrons and

$$m_i n_i \left(\frac{\partial \mathbf{U}_i}{\partial t} + \mathbf{U}_i \cdot \nabla \cdot \mathbf{U}_i \right) = -\nabla p + e n_i \left(\mathbf{E} + \mathbf{U}_i \times \mathbf{B} \right) \tag{3.4}$$

for the ions.

Assuming that the only bulk flow is due to the $\mathbf{E} \times \mathbf{B}$-drift, we have the bulk velocity in the form $\mathbf{U} = -\frac{\nabla_\perp \phi \times \mathbf{B}}{B^2}$ since the electric field is electrostatic. In the electrostatic limit the particle species are linked through Poisson's equation

$$\nabla^2 \phi = \frac{e}{\varepsilon_0}(n_e - n_i). \tag{3.5}$$

Note that $\mathbf{U}$ defined before is the same for both electron and ion guiding centres. Note also that the flow is incompressible for $\mathbf{B} = \textit{constant}$, i.e. $\nabla \cdot \mathbf{U} = 0$ since $\nabla \times (\nabla \phi \times \mathbf{B}) = 0$ here.

Taking (3.1)-(3.2), using Poisson's equation and assuming the same incompressible flow for both species, where we identify the particle position with the guiding centre position, we obtain the equation

$$\left(\frac{\partial}{\partial t} - \frac{1}{B^2}\nabla_\perp\phi \times \mathbf{B} \cdot \nabla_\perp\right)\nabla^2\phi = 0 \tag{3.6}$$

which uniquely determines the evolution of the electrostatic potential when an initial condition is given. This equation is inherently non-linear since linearisation gives trivially $\frac{\partial\phi}{\partial t} = 0$.

Taking (3.1)+(3.2) gives an equation for the evolution of the bulk plasma density

$$\left(\frac{\partial}{\partial t} - \frac{1}{B^2}\nabla_\perp\phi \times \mathbf{B} \cdot \nabla_\perp\right)n = 0 \tag{3.7}$$

with $n \equiv \frac{1}{2}(n_e + n_i)$ and ϕ is assumed given from solving 3.6. This equation determines the evolution of the entire plasma density once the potential is given.

3.1 The vortex

In fluid mechanical potential theory, a point vortex is a two-dimensional structure characterized by a velocity potential on the form

$$\Phi = \gamma \ln r \tag{3.8}$$

in two spatial dimensions. This corresponds to a so-called irrotational circulation with $\nabla \times \mathbf{U} = 0$, where the streamlines are concentric circles and the velocity is proportional to $1/r$. A line charge, i.e. an infinitely long line of charge, with constant density will have the electrostatic potential distribution

$$\phi = \frac{Q}{2\pi\varepsilon_0}\ln r. \tag{3.9}$$

By superimposing a homogeneous magnetic field $\mathbf{B}$, the $\mathbf{E} \times \mathbf{B}/B^2$ velocity in the plasma will be

$$\{U_r, U_\theta, U_z\} = \frac{Q}{2\pi\varepsilon_0 B}\left\{0, \frac{1}{r}, 0\right\} \tag{3.10}$$

in cylindrical coordinates where we introduced $\mathbf{E} = -\nabla\phi$. This potential is an exact non-linear solution of equations (3.6) and (3.7) and corresponds to "charging-up" of a magnetic field line. The result is a non-uniform rotation

of the entire plasma around the charge distribution. The angular direction of the rotation changes with the sign of the charge or the magnetic field. The resulting velocity field is equivalent to a point vortex. Throughout this thesis, such line charge structures will routinely be referred to as vortices.

By this model we represent a charge by an idealized "line" distribution. For a physically more realistic case, we have to assume a finite line width given by an average Larmor radius of the particle species in question.

3.1.1 Electron Shielding

The vortex defined above, assumes that all line charges are perfectly aligned to the magnetic field. By relaxing this assumption, and allowing perturbations to make a small angle with respect to the magnetic field, the ions will still be bound to the magnetic field, but the electrons can flow along the field lines to maintain an isothermal Boltzmann distribution $n_e(r,t) = n_0 \exp(e\phi(r,t)/\kappa T_e)$. We denote this as a "quasi two dimensional" limit. Describing the ions in two dimensions while allowing the electrons to move in this way is consistent as long as the transverse ion $\mathbf{E}_\perp \times \mathbf{B}$-velocity is much larger than the ion velocity $V_{||} \sim eE/(\omega M)$ along $\mathbf{B}$.

By using Poisson's equation on the form

$$\nabla_\perp^2 \phi = \frac{e}{\varepsilon_0}\left(\frac{en_0\phi}{\kappa T_e} - \tilde{n}_i\right) \tag{3.11}$$

electron shielding can be accounted for. Here the Boltzmann distribution is linearised and the ion density is $n_i = n_0 - \tilde{n}_i$. Combining this with the ion continuity equation and the ion velocity $-\nabla\phi \times \mathbf{B}/b^2$ gives

$$\left[\frac{\partial}{\partial t} - \frac{1}{B^2}\nabla_\perp\phi \times \mathbf{B} \cdot \nabla_\perp\right]\left(\nabla_\perp^2\phi - \frac{1}{\lambda_D^2}\phi\right) = 0 \tag{3.12}$$

which is identical to

$$\frac{\partial}{\partial t}\left(\nabla^2\phi - \frac{1}{\lambda_D^2}\right) - \frac{1}{B^2}(\nabla_\perp\phi \times \mathbf{B} \cdot \nabla_\perp)\nabla_\perp^2\phi = 0 \tag{3.13}$$

where the Debye length enters as a shielding distance.

An exact solution to this is again a line charge, but now with a different radial potential distribution. The shielded vortex will have a potential of the form $\phi(r) = aK_0(r)$ where K_0 is the modified Bessel function of the second kind.

These shielded vortices will behave differently, in that their interactions are basically limited in distance to the electron Debye length of the plasma in which they are embedded (See Section 2.2.3 for a discussion of the physical meaning of the Debye length.).

3.1.2 Some divergences

The vortex is inertia-less in the sense defined here, and we will not define a kinetic energy density associated with it, in the usual sense. (We consider an effective Hamiltonian later.) It may still be interesting to note, however that $\int_0^\infty u^2 dr \to \infty$.

3.2 Basic modes of propagation

By introducing the centre of vorticity as defined by Aref (1979) we can study the motion of a configuration of more than one vortex. The centre of vorticity is given as

$$(X, Y) = \frac{1}{\sum_\alpha \gamma_\alpha} \left(\sum_\alpha \gamma_\alpha x_\alpha, \sum_\alpha \gamma_\alpha y_\alpha \right) \tag{3.14}$$

where γ_α is the strength of vortex α, in our case $\gamma_\alpha = \frac{Q_\alpha}{2\pi\varepsilon_0}$, and (x_α, y_α) is the position of vortex α

3.2.1 One vortex

The motion of one vortex is trivial, since it can not induce motion on itself.

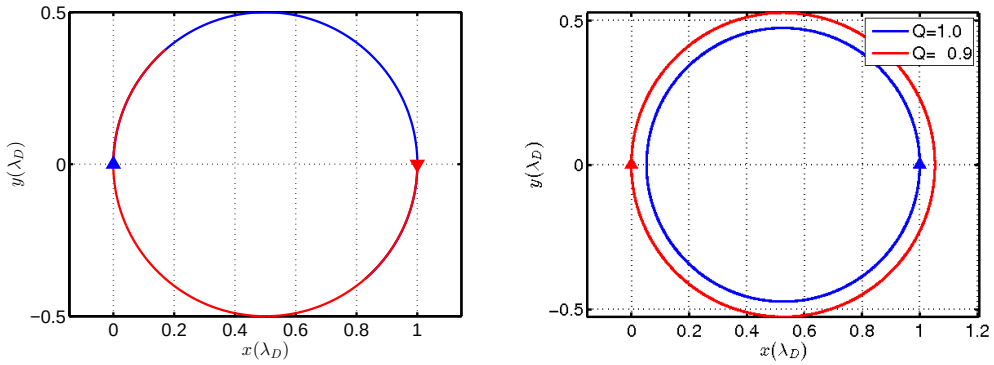
3.2.2 Two vortices

The simplest non-trivial case is to consider two vortices separated by a distance l along, say, the y-axis, with the vortices in the positions $y = \pm l/2$. Their motion can then be classified into the four different cases $Q_1 = Q_2 = Q$, $Q_1 = -Q_2$, $Q_1 > 0, \quad Q_2 > 0$ and $Q_1 > 0, \quad Q_2 < 0$ by the position of their centres of vorticity.

In the first case, the centre of vorticity will be in $(0, 0)$ and the vortices will convect each other in opposite directions, and the result will be a circular motion without net displacement. This case is illustrated in figure 3.1(a).

The second case is, arguably, more interesting. Here the centre of vorticity is in infinity, $(0, \infty)$. The vortices will then convect each other, in a way that results in a net movement in the x-direction with velocity $U = \frac{Q}{2\pi\varepsilon_0 Bl}$. Figure 3.2(a) illustrates this case

In the cases of vortices with unequal magnitudes the result will again be a net circular motion, but around a point closer to the stronger vortex, see Figure 3.1(b) and 3.2(a). In these cases the magnitude of the vorticity of the weaker vortex is 10% less than the stronger vortex. If one of the vortices is



(a) (b)

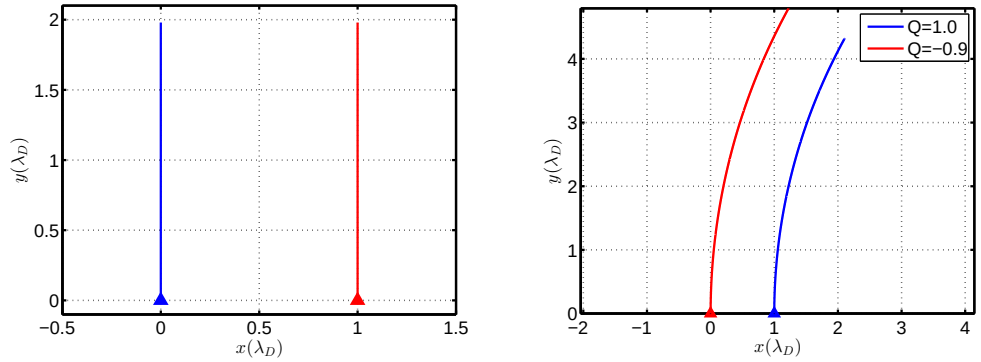
Figure 3.1: *The trajectory followed by two vortices of equal sign. (a) shows the trajectories when both vortices have the same strength* $Q = 1$ *while (b) shows the trajectories when* $Q_1 = 1.0$ *and* $Q_2 = 0.9$

much stronger than the other the result will be that the weaker orbits the stronger, with the stronger almost stationary.

3.2.3 Collision of two vortex pairs

If we arrange four vortices as shown by the arrowheads in Figure 3.3(a), with the blue arrowheads representing the vortices with positive polarities and the red representing the negative, the resulting motion of the four vortices are shown by the trajectories in Figure 3.3(a). This shows a collision of two vortex pairs; the pairs will independently move as described above when the separation between the pairs are larger than the distance between the vortices, the pairs then interact and change partners and directions.

If we shift one of the pairs a distance b in the y-direction, we call this distance the impact parameter and the case above is $b = 0$, we get different behaviour. The trajectories for different values of b is shown in figure 3.3. For low b the result is an elastic collision where the vortices change partners and direction; when b is almost equal to the distance between the vortices, the result is what could be called an inelastic collision, where the two vortices with the same polarity meet and move a circular trajectory with the other two vortices orbiting. When b is twice the separation, as shown in Figure 3.3(d) the pairs pass without much interaction and for increasing b the result approaches two independently propagating vortex pairs.



(a) (b)

Figure 3.2: *The trajectory followed by two vortices of opposite sign. (a) shows the trajectories when both vortices have the same strength* $|Q| = 1$ *while (b) shows the trajectories when* $Q_1 = 1.0$ *and* $Q_2 = -0.9$

3.2.4 Collapse of three unshielded vortices

This phenomenon will be treated with the elegant and simple formulation introduced by Kimura (1987). We now consider the (x, y)-plane as a complex plane and introduce the position of the j-th vortex as $z_j = x_j + iy_j$. The equations of motion can then be written as

$$\frac{\mathrm{d}}{\mathrm{d}t} z_j = -\frac{1}{2\pi i} \sum_{m=1}^{N}{}' \frac{\gamma_m}{\bar{z}_j - \bar{z_m}} \tag{3.15}$$

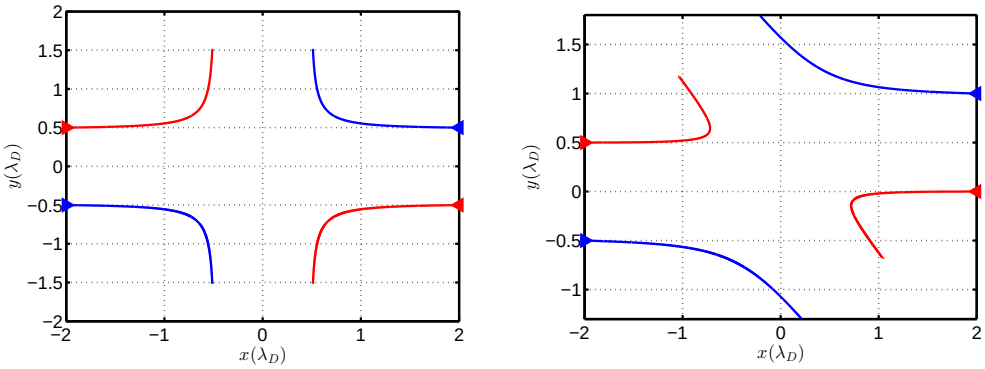
where γ_m is the strength of the j-the vortex and $\bar{z}$ is the complex conjugate of z. The prime on the summation sign means simply $j \neq m$.

In the following we postulate that a collapsing solution exists, and then demonstrate by insertion in the basic equations that the assumption was indeed correct. Thus for a collapsing triangular configuration of vortices we assume self-similar motion by $z_m = k_m f(t)$ and obtain

$$k_j \bar{f} \frac{\mathrm{d}}{\mathrm{d}t} f = i \sum_{m=1}^{N}{}' \frac{\Gamma_m}{\bar{k}_j - \bar{k}_m} \tag{3.16}$$

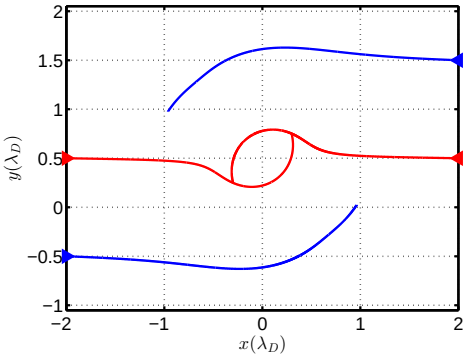
with $\Gamma_m \equiv \gamma_m / 2\pi$. We postulate that $\bar{f} \frac{\mathrm{d}}{\mathrm{d}t} f = C = A + iB$ with $A, B \in \mathbb{R}$. This reduces the problem to solving the algebraic equations

$$k_j C = i \sum_{m=1}^{N}{}' \frac{\Gamma_m}{\bar{k}_j - \bar{k}_m}, \tag{3.17}$$

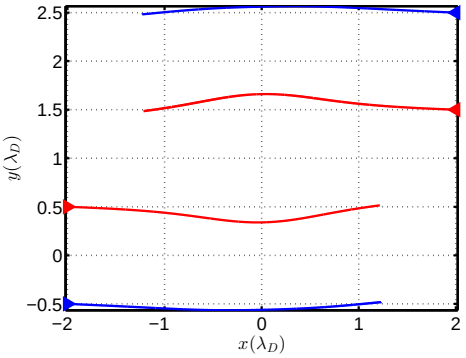


(a) $b = 0$

(b) $b = 0.5$



(c) $b = 1$



(d) $b = 2$

Figure 3.3: *Collisions of two vortex pairs with net vorticity zero for different impact parameters b. All four vortices have the same strength* $|Q| = 1$ *The blue lines in all panels show the trajectories of vortices with* $Q = +1$ *and the red lines show the trajectories of vortices with* $Q = -1$

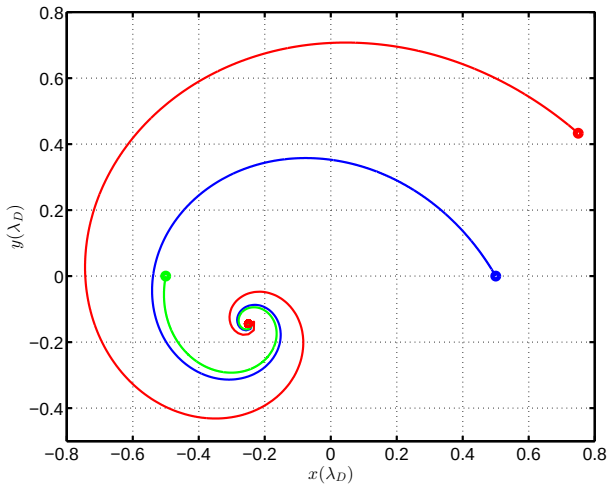


Figure 3.4: *The collapsing trajectories followed by three vortices*

which can be solved generally for $N = 3$.

The vortex collapse is algebraically unstable in the sense that a small deviation from the proper initial conditions will result in the vortices converging for some time, and then starting to diverge again. In Figure 3.4 this will happen because of numerical errors.

We can argue that a solution for three collapsing starts with a large scale, which decreases in finite time, implying a "cascade" from large to small scales. The opposite argument can be used for the inverse, explosive, case. Here we find that small scale sizes cascade to large scale.

3.2.5 Hamiltonian property of vortex systems

Before we go on to the study of vortex systems with higher numbers of vortices, it is of interest to note that a vortex system of this kind is a Hamiltonian system.

The equations of motion for point vortices can be written in the form of Hamilton's canonical equations (Aref, 1979). If the cartesian coordinates of the i-th vortex is $\mathbf{x}_i = (x_i, y_i)$, and the velocity is $\mathbf{U}_i = \left(\frac{\mathrm{d}x_i}{\mathrm{d}t}, \frac{\mathrm{d}y_i}{\mathrm{d}t}\right)$

The Hamiltonian, which accounts for the effective potential energy in the interacting vortex system, is

$$H = -\frac{1}{4\pi} \sum_{i \neq j} \gamma_i \gamma_j F(|\mathbf{r_i} - \mathbf{r_j}|), \tag{3.18}$$

where $F(|\mathbf{r}|)$ is some function accounting for the interaction between vortices and γ_i is the strength of vortex i. In the case of unshielded vortices $F(|\mathbf{r}|) = \ln(|\mathbf{r}|)$ and for the shielded vortices described in section 3.1.1, $F(|\mathbf{r}|) = K_{\mathbf{0}}(\kappa|\mathbf{r}|)$, but F can in fact be of a broad class of interactions, some of which might not be physically realizable (Lynov et al., 1991). They might, however, have the advantage of removing the singularity at $r = 0$ in the simple vortices used here.

By using the expression for the Hamiltonian (3.18) and the expression for velocity we can see that

$$\gamma_i \frac{\mathrm{d}x_i}{\mathrm{d}t} = \frac{\partial H}{\partial y_i} \tag{3.19}$$

$$\gamma_i \frac{\mathrm{d}y_i}{\mathrm{d}t} = -\frac{\partial H}{\partial x_i}, \tag{3.20}$$

which is on the same form as Hamilton's equations

$$\frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial H}{\partial q_i}, \quad \frac{\mathrm{d}q_i}{\mathrm{d}t} = \frac{\partial H}{\partial p_i}, \tag{3.21}$$

with x_i taking the role of generalised coordinate and y_i taking the role of generalised momentum. This means, as far as the hamiltonian is concerned, that one spatial coordinate is a "coordinate" and the other is a generalised momentum. This system does not contain any kinetic energy in the usual sense, since the vortices move like massless particles, instantly assuming the local flow velocity. Expression 3.18 for the Hamiltonian only applies for unbounded systems. The presence of periodic boundary conditions break rotational symmetry, and the potential will have to be modified.

Since the Hamiltonian does not depend explicitly on time, it is an integral of motion. The Hamiltonian is, as well, invariant to translations and rotations of the coordinates, this implies three conservation laws.

$$\sum_i \gamma_i x_i = constant \tag{3.22}$$

$$\sum_i \gamma_i y_i = constant \tag{3.23}$$

$$\sum_i \gamma_i (x_i^2 + y_i^2) = constant. \tag{3.24}$$

If the sum of all vortex strengths is non-zero, $\sum_i \gamma_i \neq 0$, the centre of vorticity is a fixed point for the flow with coordinates

$$\{X, Y\} \equiv \frac{1}{\sum_i \gamma_i} \left\{ \sum_i \gamma_i x_i, \sum_i \gamma_i y_i \right\}. \tag{3.25}$$

By introducing the vortex separation as $l_{i,j} \equiv |\mathbf{r_i} - \mathbf{r_j}|$, it can be shown that

$$\frac{1}{2}\sum_i \gamma_i\gamma_j l_{i,j}^2 = \left(\sum_i \gamma_i\right)\sum_j \gamma_j(x_j^2 + y_j^2) - \left(\sum_i \gamma_i x_i\right)^2 - \left(\sum_i \gamma_i x_i\right)^2, \tag{3.26}$$

is a constant of motion as well, independent of the reference system. From equations (3.19) it is possible to derive an expression for the square of the vortex separation without reference to the absolute positions of the vortices. For unshielded vortices we have

$$\frac{\mathrm{d}}{\mathrm{d}t} l_{i,j}^2 = \frac{2}{\pi}\sum_{k\neq i, k\neq j} \gamma_k \sigma_{i,j,k} A_{i,j,k}\left(\frac{1}{l_{j,k}^2} - \frac{1}{l_{k,i}^2}\right), \tag{3.27}$$

where the sum excludes $k = i$ and $k = j$. The quantity $\sigma_{i,j,k}$ defines the orientation of the triangle spanned by three vortices i, j, k so that $\sigma_{i,j,k} = +1$ if i, j, k appear in counter clockwise order, and $\sigma_{i,j,k} = -1$ otherwise. $A_{i,j,k}$ is the area of the triangle. Note that $\sigma_{i,j,k}$ is undefined if the three vortices are on a line, but that in this case the area is zero.

The relations (3.25) together with the Hamiltonian (3.18) and the constant of motion (3.26) defines the problem of the motion of N vortices. Equation (3.25) indicates that the problem of $N = 3$ is fundamental for the understanding of $N > 3$, since 3 is the lowest number of vortices where the right hand side of (3.25) is nonzero, and the separation of the vortices is not a constant of motion. This means that if we consider the separations as quantitative measures of excited scales of motion, the three-vortex interaction is the lowest order interaction capable of exciting new scales. The three vortex problem is Hamiltonian with six degrees of freedom and four constants of motion with the vortex strengths as constant parameters, and as such is integrable as shown by Poincaré (1893) (See Aref (1983) for a historical review of the research on vortex dynamics.). Some of the solutions of this motion is highly unexpected and surprising, so a summary of these special results will be given before embarking on the treatment of N vortices.

3.3 Many vortices

3.3.1 Many vortices with deterministic positions

For cases with more than two vortices, it is possible to construct stable patterns of vortices. See for example (Morikawa and Swenson, 1971) for a

discussion about the linear and non-linear stability of some such patterns of geostrophic vortices.

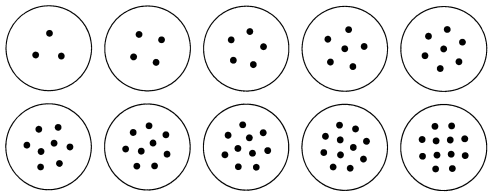


Figure 3.5: *Illustration of stationary line vortex distributions. The circles drawn around each figure are inserted merely to separate different solutions. Figure provided by Mitsuo Kono.*

Consider the model equation

$$\frac{\partial}{\partial t}(\kappa^2\phi - \nabla^2\phi) - \{\phi; \nabla^2\phi\} = 0, \tag{3.28}$$

which includes both shielded and unshielded cases for $\kappa = 1$ and $\kappa = 0$ respectively, and where $\{;\}$ is the Poisson brackets, defined as

$$\{a(x,y); b(x,y)\} \equiv \frac{\partial a}{\partial x}\frac{\partial b}{\partial y} - \frac{\partial b}{\partial x}\frac{\partial a}{\partial y} \equiv \hat{\mathbf{z}} \cdot \nabla a \times \nabla b. \tag{3.29}$$

Point vortices can be introduced through

$$(\kappa^2 - \nabla^2)\phi(\mathbf{r}, t) = \sum_\alpha \gamma_\alpha \delta(\mathbf{r} - \mathbf{r}_\alpha). \tag{3.30}$$

Inserting (3.30) into (3.28) gives the vortex potential as

$$\phi(\mathbf{r}, t) = \sum_\alpha \frac{\gamma_\alpha}{2\pi} K_0(\kappa|\mathbf{r} - \mathbf{r}_\alpha(t)|), \tag{3.31}$$

where K_0 is the modified Bessel function of the second kind of order zero. If $\kappa \to 0$ the potential becomes

$$\phi(\mathbf{r}, t) = \sum_\alpha \frac{\gamma_\alpha}{2\pi} \ln(|\mathbf{r} - \mathbf{r}_\alpha(t)|), \tag{3.32}$$

which is a solution for unshielded point vortices, and essentially a sum of potentials on the form of (3.9).

The equations of motion for the vortices are

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{r}_\alpha &= \hat{\mathbf{z}} \times \nabla\phi(\mathbf{r}_\alpha, t) \\ &= \frac{\kappa}{2\pi}\sum_\beta \gamma_\beta \frac{\hat{\mathbf{z}} \times (\mathbf{r}_\alpha - \mathbf{r}_\beta)}{|\mathbf{r}_\alpha - \mathbf{r}_\beta|} K_1(\kappa|\mathbf{r}_\alpha - \mathbf{r}_\beta|),\end{aligned} \tag{3.33}$$

where the K_1-function arises from the differentiation of K_0. These equations mean, physically, that each vortex, at any time, is convected by the superimposed velocity field from all the other vortices. When the distances between vortices is short enough, compared to the shielding distance, shielding can be neglected, and the equation (3.34) simplifies to

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{r}_\alpha = \frac{1}{2\pi}\sum_\beta \gamma_\beta \frac{\hat{\mathbf{z}} \times (\mathbf{r}_\alpha - \mathbf{r}_\beta)}{|\mathbf{r}_\alpha - \mathbf{r}_\beta|^2} \tag{3.34}$$

for $\kappa \to 0$, which could also have been derived from equation (3.32)

Equilibrium configurations have been discussed by Morikawa and Swenson (1971) for N vortices equally distributed on a circle without a centre vortex, see e.g. Figure 3.5. They found that for $2 \leq N \leq 6$ the configuration is stable for $\kappa < 1.289$ and that $N = 7$ is stable for $\kappa = 0$. When a centre vortex is present the configuration $N = 3$ is stable for $\kappa \geq 3.50$, configurations of $N = 4, 5, 6, 7$ vortices are stable for all κ, for $N = 8$ the configuration is stable for $\kappa < 1.597$ and $N = 9$ gives a stable configuration only for $\kappa = 0$.

A more general analysis Kono et al. (1998) can be made by introducing the distribution function

$$f(\mathbf{r}, t) = \sum_\alpha \gamma_\alpha \delta(\mathbf{r} - \mathbf{r}_\alpha(t)) \tag{3.35}$$

and rewriting equation 3.34 as

$$\frac{\partial}{\partial t} f(\mathbf{r}, t) = -\frac{\kappa}{2\pi}\int d\mathbf{r}' \frac{K_1(\kappa|\mathbf{r} - \mathbf{r}'|)}{|\mathbf{r} - \mathbf{r}'|}\hat{\mathbf{z}} \times (\mathbf{r} - \mathbf{r}') \cdot \nabla f(\mathbf{r}', t) f(\mathbf{r}, t) \tag{3.36}$$

Changing coordinates from Cartesian to cylindrical (r, θ) and remembering that

$$\hat{\mathbf{z}} \times (\mathbf{r} - \mathbf{r}') \cdot \nabla = -r' \sin(\theta - \theta')\frac{\partial}{\partial r} + \left[1 - \frac{r'}{r}\cos(\theta - \theta')\right]\frac{\partial}{\partial \theta}, \tag{3.37}$$

we get

$$\frac{\partial}{\partial t} f(r, \theta, t) + \Omega(r, \theta, t)\frac{\partial}{\partial \theta} f(r, \theta, t) + V(r, \theta, t)\frac{\partial}{\partial r} f(r, \theta, t) = 0, \tag{3.38}$$

where

$$\Omega(r,\theta,t) = -\frac{\kappa}{2\pi}\int d\mathbf{r}' \frac{K_1(\kappa|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|}\left[1-\frac{r'}{r}\cos(\theta-\theta')\right] f(r,\theta,t) \tag{3.39}$$

$$V(r,\theta,t) = -\frac{\kappa}{2\pi}\int d\mathbf{r}' r' \frac{K_1(\kappa|\mathbf{r}-\mathbf{r}'|)}{|\mathbf{r}-\mathbf{r}'|}\sin(\theta-\theta') f(r,\theta,t) \tag{3.40}$$

From equations (3.39) and (3.40) equilibrium configurations are obtained by imposing that $V(r,\theta,t)$ has to be identically zero and that $\Omega(r,\theta,t)$ has to be constant. Under these conditions the system is rigidly rotating around the origin without any radial displacement

It is, using the foregoing equations, possible to look for stable concentric structures of M rings with radius a_m with N_m identical vortices equally distributed on the m-th ring ($m = 1, ..., M$). This means that $\gamma_\alpha = \gamma$. The pattern can be with or without a center vortex γ_0 which corresponds to $\gamma_0 = \gamma$ or $\gamma_0 = 0$. Then

$$f(r,\theta,t) = \sum_{m,n}\frac{\gamma}{a_m}\delta(r-a_m)\delta(\theta-\theta_{m,n}) + \frac{\gamma_0}{r}\delta(r)\delta(\theta) \tag{3.41}$$

for which the following holds in equilibriun

$$V(m_0,n_0) = -\frac{\kappa^2}{2\pi}\gamma\sum_{m,n}\frac{a_m}{\sigma_{m,n}}K_1(\sigma_{m,n})\sin(\theta_{m_0,n_0}-\theta_{m,n}) = 0 \tag{3.42}$$

and

$$\Omega(m_0,n_0) = -\frac{\kappa^2}{2\pi}\gamma\sum_{m,n}\frac{1}{\sigma_{m,n}}K_1(\sigma_{m,n})\left[1-\frac{a_m}{a_{m_0}}\cos(\theta_{m_0,n_0}-\theta_{m,n})\right] + \frac{\kappa}{2\pi}\gamma_0\frac{K_1(\kappa a_{m_0})}{a_{m_0}} = \text{const}, \tag{3.43}$$

where

$$\sigma_{m,n} = \kappa\sqrt{a_{m_0}^2 + a_m^2 - 2a_{m_0}a_m\cos(\theta_{m_0,n_0}-\theta m,n)} \tag{3.44}$$

and (m,n) refers to the n-th vortex on the m-th ring and (m_0,n_0) refers to the coordinates of an arbitrary reference vortex.

If the vortices are distributed on each circle, in such a way that that the distances between neighbouring vortices is the same, as illustrated in Figure 3.5, that is $\theta_{m,n} = 2\pi n/N_m$ where N_m is the number of vortices on the m-th circle, then $V(m_0,n_0)$ is always equal to zero and $\Omega(m_0,n_0)$ does not depend on n_0 because of symmetry.

3.4 Randomly distributed ensemble of vortices

3.4.1 Localised cloud

The dynamics of a localised cloud will depend on the sign of the vorticities. If all vortices have the same polarity, the dominating motion will be on of rotation, since the total vorticity is large. The Hamiltonian of this system will be large and positive. The cloud will rotate almost as a rigid body and we expect that the cloud will preserve it's identity and structure over time.

A cloud with approximately zero total vorticity will have no net rotation, and we will expect that vortices with opposite polarities will pair up and propagate away from the cloud. This system will have a small Hamiltonian since pairs of $++, +-, -+, --$ have equal probability. We do not expect this system to preserve it's structure over time.

3.4.2 Homogeneous ensembles

With an infinite system, we mean one which is homogeneous, isotropic and in equilibrium, so, apart form random fluctuations, the quantities are time stationary. The transport properties of such an infinite system will be treated as those arising from a random turbulent velocity field, and discussed in detail in Chapter 4.

The realisation of such a system for numerical simulation is treated in Chapter 5.

3.5 Negative temperatures

In standard applications of classical statistical mechanics the idea of negative temperatures might seem to be leading to paradoxes. It might be worthwhile to give a simple illustrative summary demonstrating the properties of realizable physical systems which can logically be assigned a negative temperature (Marvan, 1966). Such systems consisting of simple particle spins are to some extent simpler to analyse than a collection of line vortices.

Standard and intuitive objections to the concept of negative temperatures are often based on models derived from ideal gases, and indeed it will not be possible to give a meaningful definition of negative temperatures in that context. We heat a gas by supplying energy: the more energy we add, the warmer the gas becomes. On the other hand we will not be able to cool the gas below the absolute zero. Note, however, that although energy

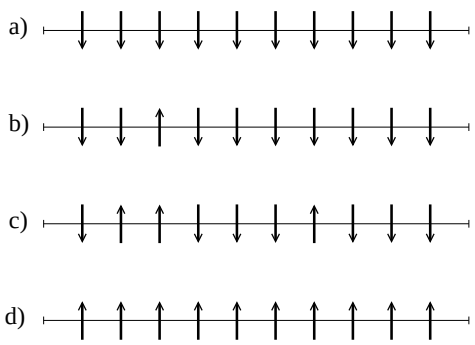


Figure 3.6: *Illustration of a simple spin system with two states,* ± 1 *with the lowest energy state being one where all spins are* -1 *as in a). The number of possible microstates is* $w = N!/(m!(N-m)!)$*, when* m *is the number spins with* $+1$*. For the case shown we have* $N = 10$*.*

and temperature are indeed related, the relation is different for different physical systems. Consider for instance a simple system with N-spins having only two discrete states, one negative and one positive assigned the values -1 and $+1$ without loss of generality, see also Figure 3.6. If the system is subject to an externally imposed homogeneous magnetic field we will argue that the lowest energy state is one where all spins take the value -1. If we now supply a certain minimum energy to the system we might flip one of the spins to $+1$, see Figure 3.6b). Adding more energy we might flip one more, etc. Eventually, we will, however, find that all spins are in the state $+1$ and no more energy can be accommodated in the system. It could be argued that the lowest state is assigned the temperature $T = 0$, while the final state has $T = T_{max}$. This definition is, however, in variance with another intuitive interpretation of temperatures, namely that particles in hot systems are more randomly distributed that in cold systems. The classical measure of randomness in statistical mechanics is entropy. Some textbooks, Pathria (1998) for instance, actually use entropy as a basis for discussing statistical mechanics. Randomness is a concept we associate with lack of predictability, and it is indeed this property that is used when applying the ideas to communication theory when introducing the "Shannon entropy". To sort out these different concepts it might be an advantage to give short summary.

Disorder is a manifestation of the largeness of the number of microstates the system can have with equal probability. The larger the choice of microstates, the lesser the degree of predictability or the level of order in the system. Complete order prevails when and only when the system has no

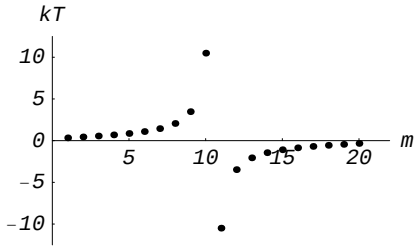


Figure 3.7: *Temperature of a spin system as illustrated in Figure 3.6, here with $N = 20$. Starting from the minimum state, as in Figure 3.6a) and adding energy to flip the spins one by one, we see the temperature increasing, but after reaching the state with maximum number of micro-states we obtain negative temperatures. Figures like this were for a time used to argue that negative temperatures were larger than infinity!*

choice but to be in a unique state: this, in turn, corresponds to the state of vanishing entropy (Pathria, 1998). Taking our example in Figure 3.6, we find that we have only one possible state for all spins +1 or -1. All spins with the exception one being -1 have N possible microstates, where N is the number of spins. With all spins with the exception two being -1 have $N(N-1)$ possible microstates. We will denote the number of ways a microstate can be realized by the letter w, the thermodynamic weight. For m positive spins we have in our case $w = N!/(m!(N-m)!)$ possible microstates, where the maximum value is reached for $m = N/2$, or its closest integer value. In place of the thermodynamic weight, we frequently use the entropy, $S \equiv k \ln w$, where k is a universal constant independent of the substance considered.

In general, temperature is a measure of change of entropy corresponding to a unit change of energy, ΔU. The example illustrated in Figure 3.6 was used to demonstrate how the disorder changed by adding energy to the system, but also how the system became ordered again, when we exceeded the maximum number of available microstates. We can also argue that temperature is a measure change of disorder in a system resulting from a unit change of energy, $\Delta S/\Delta U = 1/T$. Again for the example in Figure 3.6 we have $S = k \ln(N!/(m!(N-m)!))$, i.e. $1/T = \Delta S/\Delta m = k \ln(((m+1)!(N-m-1)!)/(m!(N-m)!))$ with $\Delta m = 1$ for the flipping on one spin in Figure 3.6. The result is illustrated in Figure 3.7 for $N = 20$.

In general it can be stated that negative temperatures are always associated with systems in which the higher energy levels are more densely occupied than the lower levels. Matter in maser and laser state are associated with negative temperatures.

3.5.1 Negative temperature states

In certain limits it is meaningful to assign negative temperatures to a collection of vortices, as illustrated in Figure 3.8. In principle, the discussion applies for both shielded and unshielded vortices. For the shielded case we will require that the vortices forming the pairs in Figure 3.8a) are placed within a shielding distance.

By the Hamiltonian property of vortex systems we have that $H < 0$ for a system illustrated in Figure 3.8a) where positive and negative vortices are close in pairs, while $H > 0$ for the system shown in Figure 3.8b) where positive and negative vortices are clumped together in separate groups. To get from the state given in Figure 3.8a) to the one shown in Figure 3.8b) we have to supply energy. We thus start with an ordered system consisting of many vortex pairs and end with another organized system where vortices are grouped according to their sign. In between for some $H \approx 0$ we have a state of maximum randomness, where there is an approximately equal probability of finding a negative or a positive vortex in the vicinity of any chosen vortex.

With a simple spin-system it is easy to provide a measure for the randomness or disorder of the system: for the present case a complete order is an extreme and it is preferable to find a means for illustrating local order. A modified version of the structure function known in classical turbulence studies serves such a purpose. We thus place the origin of a reference system at a local maximum of the vorticity associated with a vortex and plot the distance to the most nearby vortex, multiplying the signs of the two vortices. Then we continue to the next vortex and repeat for some prescribed distance. This procedure is repeated by placing the origin at a different reference vortex, and this way an ensemble of realizations can be built up. For high local order we will find the sign of the average structure function obtained by this procedure to be either large and negative (when a nearby vortex always have the opposite sign of the reference vortex and $H > 0$) or large and positive (when a nearby vortex always have the same sign of the reference vortex and $H < 0$). Recall the minus sign in the definition of H in (3.18). For the case of large positive H we expect a short correlation length, of the order of the average relative distance between the positive and negative vortices forming the vortex-pairs in Figure 3.8a). For the case of large negative H, we expect a long correlation distance being of the order of the average size of the vortex "clouds" illustrated in Figure 3.8b).

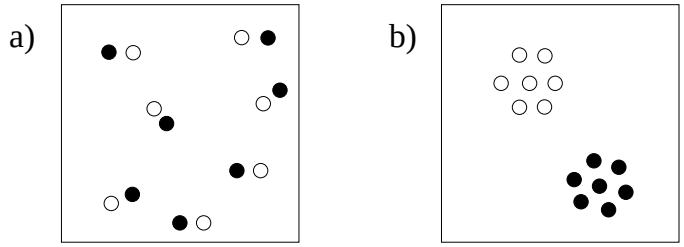


Figure 3.8: *Illustration of two vortex states: one with a large positive energy in a) and one with a large negative energy in b).*

Chapter 4

Turbulent diffusion and transport

Diffusion is an important transport phenomenon in many physical systems. Classical diffusion can be derived from simple random walk processes and will usually be slow. In many plasma physics systems, the transport is many times higher than this classical diffusion (Tennekes and Lumley, 1972). These systems are often turbulent and one of the most important properties of turbulent flows is their ability to disperse particles at a rate which by far exceeds transport by classical molecular diffusion (Tennekes and Lumley, 1972). This property is the same for neutral fluids and for plasmas.

We will make some general remarks on classical diffusion and Brownian motion, and we will present some basic results on turbulent diffusion.

4.1 Classical diffusion

Classical diffusion can be derived from the continuity equation (see section 2.3.3 and (3.1), (3.2)) on the form

$$\frac{\partial \phi(\mathbf{r},t)}{\partial t} + \nabla \cdot \mathbf{j} = 0 \tag{4.1}$$

where $\phi(\mathbf{r},t)$ is the density of the diffusing material at point $\mathbf{r}$ and time t, and $\mathbf{j}$ is the flux of the diffusing material. ϕ can represent any scalar quantity, also temperature. No material is created or destroyed. By postulating that the driving force of the flux is a gradient in ϕ and that this force is proportional to $\nabla\phi$, this is called Fick's first law[1], i.e. $\mathbf{j} = -D(\phi)\nabla\phi(\mathbf{r},t)$ we arrive at the

[1]This is a phenomenological law based on the experimentally confirmed assumption that macroscopic fluxes originate from density gradients (Rumer and Ryvkin, 1980).

diffusion equation.

$$\frac{\partial \phi(\mathbf{r},t)}{\partial t} = \nabla \cdot [D(\phi)\nabla\phi(\mathbf{r},t)] \tag{4.2}$$

which simplifies to the linear diffusion equation if $D(\phi) = D = constant$

$$\frac{\partial \phi}{\partial t} = D\nabla^2\phi. \tag{4.3}$$

4.1.1 Diffusion of a light particle

Consider a light point charge which is free to move due to collisions with other particles in the environment. The displacement of this particle can be analysed quite simply (Pécseli, 2000). The particle will have constant velocity w between collisions, so it's displacement from z_0 in the z-direction is $z = z_0 + wt$ until the next collision. The displacement relative to z_0 can be written on the form

$$z = \int_0^{t_1} w_1 dt + \int_{t_1}^{t_2} w_2 dt + ... + \int_{t_{N_1}}^{\mathcal{T}} w_N dt = \sum_{r=1}^{N} w_r(t_{r-1} - t_r) \tag{4.4}$$

and similarly for the other coordinates. We assume that the velocity of the particle after each collision is random with $\langle w_r \rangle = 0$ and independent of the velocity before the collision. Then $\langle z \rangle = 0$ is trivial. To get a non-trivial result we consider $\langle z^2 \rangle$

$$\langle z^2 \rangle = \sum_{s=1}^{N}\sum_{r=1}^{n} \langle w_s w_r (t_{s-1} - t_s)(t_{r-1}t_r)\rangle. \tag{4.5}$$

If we now consider a very short time interval $\mathcal{T}$ in which collisions can be ignored, we get $z = \int_0^{\mathcal{T}} w_1 dt = w_1\mathcal{T}$ and

$$\langle z^2 \rangle = w_1^2\mathcal{T}^2 \tag{4.6}$$

This short time limit is determined purely by inertial effects and is independent on collision frequency.

Now we consider the particle after it has undergone many collisions. Since the velocities in different time intervals are statistically independent and also

independent of the particular time interval, we get

$$\begin{aligned}\langle z^2\rangle &= \sum_{s=1}^{N}\sum_{r=1}^{n}\langle w_s w_r (t_{s-1}-t_s)(t_{r-1}t_r)\rangle \\ &= \langle w^2\rangle \sum_{s=1}^{N}\langle (t_{s-1}-t_s)^2\rangle \\ &= 2\frac{\kappa T}{m}\frac{N}{\nu^2},\end{aligned}$$

where ν is the particle collision frequency. We assumed that $\langle w_s w_r\rangle = \langle w_s^2\rangle\delta_{r,s} = \langle w^2\rangle\delta_{r,s}$ in terms of the Kronecker delta, since the particle's velocity after the collision is considered completely uncorrelated to the velocity before the collision. (This assumption is valid for light particles, but not for heavy particles moving against a background of light particles.) We used $\langle (t_{s-1}-t_s)^2\rangle = 1/\nu^2$ for all s. With the assumption that the particle being in thermal equilibrium after one collision, we also used $\langle w^2\rangle = \kappa T/m$. The number of collisions, N, in the series varies statistically from realization to realization and, averaging over the realizations, we have $\langle N\rangle = \mathcal{T}\nu$. This gives

$$\langle z^2\rangle = 2\frac{\kappa T}{m}\frac{\mathcal{T}}{\nu}. \tag{4.7}$$

This limit for large $\mathcal{T}$ is often called the diffusion limit with reference to the solutions of the standard diffusion equation; see Section 4.1. In this case the diffusing material would be the probability density $P(z,t)$ for the particles displacement z. The solution to this diffusion equation is $P(z,t) = \sqrt{4\pi Dt}\exp[-z^2/(4Dt)]$ if the particle was at $z=0$ at $t=0$, i.e. $P(z,t=0) = \delta(z)$. In this case we get $\langle z^2\rangle = \int_{-\infty}^{\infty} z^2 P(t,z) = 2Dt$ which is analogous to (4.7) if we define

$$D = \frac{\langle w^2\rangle}{\nu} = \frac{\kappa T}{m\nu}. \tag{4.8}$$

4.2 Turbulent diffusion

Turbulent diffusion is manifestly different form classical Brownian diffusion. This is best seen by considering a cloud of particles released at the same time, see Figure 4.1. Turbulent diffusion moves and distorts the entire cloud, but in the asymptotic diffusion limit ($t\to\infty$), the two figures (A) and (B) tends to become similar.

We will now show some of the basic results on turbulent diffusion. The outline is independent of the dimensionality of the problem and the ideas

apply equally well for three dimensional and two dimensional turbulence, given the assumption of homogeneity and isotropy. Particle displacement will be expressed in terms of a velocity, but this velocity can be obtained from an electrostatic electric field in magnetized plasmas as $\mathbf{u}(\mathbf{r},t) = \mathbf{E}(\mathbf{r},t) \times \mathbf{B}_0/B_0^2$ for homogeneous magnetic fields, assuming low frequency turbulence where all relevant frequencies are $\omega \ll \Omega_{ci}$. This model will then assume two dimensional turbulence in the plane $\perp \mathbf{B}_0$. Many experiments have demonstrated that turbulent transport is an important mechanism in for instance fusion plasma experiments (Liewer, 1985), and significant efforts have been made to understand this mechanism for anomalous transport also in the context of drift wave turbulence (Horton, 1999).

4.2.1 Single particle diffusion

Let us again consider the simplest possible problem, namely the one where a single particle is released in a homogeneous and isotropic turbulent velocity field, $\mathbf{u}(\mathbf{r},t)$; the field generated from an ensemble of vortices is such a velocity field. We assume that the particle is convected as a passive scalar by the flow, and want to determine its mean-square displacement with respect to the origin of release, which we without loss of generality take to be at the origin of the coordinate system (Roberts, 1957).

Since, by assumption, $d\mathbf{r}(t)/dt = \mathbf{u}(\mathbf{r}(t),t)$, we have in a given realization of the flow the particle position to be

$$\mathbf{r}(t) = \int_0^t \mathbf{u}(\mathbf{r}(t'),t')dt' \,. \tag{4.9}$$

The *average* position $\langle \mathbf{r}(t) \rangle$ vanishes, since we have assumed $\langle \mathbf{u}(\mathbf{r}(t),t) \rangle = 0$. This is actually not quite as self evident as it might appear (Tennekes and Lumley, 1972), since we can assume from the outset only that $\langle \mathbf{u}(\mathbf{r},t) \rangle = 0$, but this is concerning a function of a spatial as well as a temporal variable, while $\langle \mathbf{u}(\mathbf{r}(t),t) \rangle$ is a function of time alone. We postpone the discussion of this question.

The mean square displacement is a positive quantity, and we find

$$\begin{aligned} \mathbf{r}(t) \cdot \frac{d\mathbf{r}(t)}{dt} &= \frac{1}{2}\frac{dr^2(t)}{dt} = \mathbf{u}(\mathbf{r}(t),t) \cdot \int_0^t \mathbf{u}(\mathbf{r}(t'),t')dt' \\ &= \int_0^t \mathbf{u}(\mathbf{r}(t),t) \cdot \mathbf{u}(\mathbf{r}(t'),t')dt' \,. \end{aligned} \tag{4.10}$$

Taking the ensemble average, we have

$$\frac{d}{dt}\left\langle r^2(t) \right\rangle = 2\int_0^t \left\langle \mathbf{u}(\mathbf{r}(t),t) \cdot \mathbf{u}(\mathbf{r}(t'),t') \right\rangle dt' \,. \tag{4.11}$$

For time stationary turbulence we require $\langle \mathbf{u}(\mathbf{r}(t),t)\cdot\mathbf{u}(\mathbf{r}(t'),t')\rangle = R_L(t-t')\langle u^2\rangle$, with the subscript L reminding us to sample the velocity field along a Lagrangian orbit. We introduced R_L as the normalized Lagrangian velocity correlation function, $R_L(0)=1$.

When integrating (4.10) as $\langle r^2(t)\rangle = 2\langle u^2\rangle \int_0^t \int_0^{t''} R_L(t''-t')dt'dt''$, it is an advantage to introduce the variables $\tau \equiv t''-t'$ and $s\equiv t''$, giving the Jacobian

$$J = \left| \begin{array}{cc} \dfrac{\partial\tau}{\partial t''} & \dfrac{\partial\tau}{\partial t'} \\ \dfrac{\partial s}{\partial t''} & \dfrac{\partial s}{\partial t'} \end{array} \right| = \left| \begin{array}{cc} 1 & -1 \\ 1 & 0 \end{array} \right| = 1\,,$$

so that $dt''dt' = d\tau ds$. Making the variable transforms as indicated, we note that we can change the order of integration as $\int_0^t\int_0^s R_L(\tau)d\tau ds = \int_0^t\int_\tau^t R_L(\tau)dsd\tau$ to give

$$\left\langle r^2(t)\right\rangle = 2t\langle u^2\rangle \int_0^t (1-\tau/t)R_L(\tau)d\tau\,. \tag{4.12}$$

Two relevant limiting cases of (4.12) can be distinguished here (Roberts, 1957).

1. Very short times, where it can be assumed that $R_L(\tau)\approx 1$. In this limit we find $\langle \mathbf{r}^2(t)\rangle \approx \langle u^2\rangle t^2$, which is often called the *ballistic limit* since it is the result we would have obtained by assuming the particle to follow straight lines of orbit, $\mathbf{r}(t)=\mathbf{u}t$, and simply average over all velocities. We might experience to find auto-correlation functions that are not differentiable for $\tau=0$, and in such cases the ballistic limit does not exist. Such a case requires that $\langle(d\mathbf{u}/dt)^2\rangle$ diverges. With finite particle inertia we would expect that $d\mathbf{u}/dt$ should be finite at all times. The absence of a ballistic limit simply indicates a very rapidly changing velocity field, and the interval where $R_L(\tau)\approx 1$ may be negligible.

2. Very large times, $t\to\infty$, where it can be assumed that $\int_0^t R_L(\tau)d\tau \approx \tau_L$, introducing the Lagrangian integral correlation time

$$\tau_L \equiv \int_0^\infty R_L(\tau)d\tau\,. \tag{4.13}$$

In this limit we find the important result

$$\left\langle r^2(t)\right\rangle \approx 2t\tau_L\langle u^2\rangle\,. \tag{4.14}$$

At least formally this looks like the result one obtains by using the classical diffusion equation with a diffusion coefficient $D \equiv 2\tau_L \langle u^2 \rangle$ to obtain the mean square particle displacement. Such cases have $\langle \mathbf{r}^2(t) \rangle \sim t$. The limit of times much larger than the Lagrangian correlation time is consequently called the diffusion limit. This limiting case is consistent with a random walk with a typical length step $\tau_L \sqrt{\langle u^2 \rangle}$ per time τ_L.

Note that $\mathbf{r}(t)$ is not a time-stationary random process, although it is derived from $\mathbf{u}(\mathbf{r}, t)$ which can be assumed to be so! The initial time (the time of release of the particle) has a special role for $\mathbf{r}(t)$.

Introducing the Lagrangian power spectrum,

$$S_L(\omega) \equiv (1/2\pi) \int_{-\infty}^{\infty} R_L(t) e^{-i\omega t} dt$$

, with $R_L(t) = \int_{-\infty}^{\infty} S_L(\omega) e^{i\omega t} d\omega$, we can write the result (4.12) on the form (Lumley and Panofsky, 1964)

$$\left\langle r^2(t) \right\rangle = t^2 \left\langle u^2 \right\rangle \int_{-\infty}^{\infty} \left(\frac{\sin(\omega t/2)}{\omega t/2} \right)^2 S_L(\omega) d\omega \,. \tag{4.15}$$

The function $\sin^2(\omega t/2)/(\omega t/2)^2$ originates from the Fourier transform of the "triangular" function $1 - \tau/t$ entering the convolution (4.12). For large times it is evident that dispersion of the test particle is primarily due to the low frequencies in the Lagrangian spectrum. We have $\lim_{t\to\infty} \sin(\omega t/2)/(\omega/2) = \pi\delta(\omega)$, so we recover $\tau_L = S_L(\omega = 0)$. Note that oscillations in the Lagrangian spectrum with frequencies being multipla of $1/t$ do not contribute to the particle displacement. This is because they return the particle to its starting point after a time t (Batchelor, 1949). Often it is assumed that low frequencies in (4.15) corresponds to large wavelengths (or rather large scales), but we should be aware that there is no a priori reason to believe this.

By (4.15) our problem seems to be solved once and for all, at least for the case of homogeneous isotropic turbulence! Alas, it is not so, since we do not know the spectrum $S_L(\omega)$, and it is very complicated to obtain it experimentally. It has been a major enterprise over the years to find ways of predicting $S_L(\omega)$ on the basis of the more readily measurable *Eulerian* correlation function. Amazingly good results can be obtained, at least as far as predictions of $\langle \mathbf{r}^2(t) \rangle$ are concerned. However, this might as well imply that this is a very robust results, and that almost any reasonable guess on R_L will give acceptable results. After all, we *must* require that $R_L(0) = 1$ and that $R_L(t \to \infty) \to 0$, and with a little common sense all reasonable guesses of R_L tend to look more or less the same.

Figure 4.1: *The motion of a set of particles over time when we have (A) simple brownian motion and (B) when there is turbulent diffusion. The figure is originally produced by Jakob Mann and Søren Ott at the Risø National Laboratory.*

The problem of relating the Eulerian and Lagrangian description in general is an extremely complex one, and no simple answers exists (Lumley, 1962). Corrsin (1963) has shown with dimensional reasoning that for extremely large Reynolds numbers τ_L is roughly equal to the Eulerian integral time scale τ_E. This equality is somewhat at variance with intuition, since the Eulerian time correlation correlates at every instant new fluid at the observation point, whereas the Lagrangian correlation deals always with the same fluid. Therefore one would expect that the Lagrangian time scale would be no larger than the Eulerian one, and generally larger (Lumley and Panofsky, 1964). Semmingly there are equally good arguments for predicting $\tau_L > \tau_E$ as wella as $\tau_L < \tau_E$.

When doing numerical simulation, like we have done here, it is simple to obtain both the Lagrangian and the Eulerian correlation functions, since we can follow the trajectories of single particles, and sample the field at selected points. This means that numerical simulation is one of the techniques that can be used to investigate the relationship between Lagrangian and Eulerian descriptions of a system. If an analytical model for such a relation tests favourably in a simple two dimensional simulation, it can be applied with some confidence also for a three dimensional simulation.

4.2.2 Eulerian and Lagrangian mean-square velocities

Before closing the present discussion, we should clear up an ambiguity hinted already: we introduced $\langle u^2 \rangle$ in a Lagrangian context: we should in principle distinguish ensemble averages of Eulerian and a Lagrangian velocities. These two are however simply related in cases relevant here. Consider first the Eulerian characteristic function for the fluctuating velocity which we can define as the ensemble average $\langle \exp(i\mathbf{k} \cdot \mathbf{u}_E(\mathbf{r}, t) \rangle$. The corresponding structure function for the Lagrangian velocity field, as sampled by a self consistently moving passive test particle is $\langle \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t) \rangle$. We then integrate the two exponential functions entering the characteristic functions over a (large) volume, V. Selecting a certain reference time, the volume is defined by the same boundaries for both integrals, i.e. we have

$$\frac{1}{V} \int_V \exp(i\mathbf{k} \cdot \mathbf{u}_E(\mathbf{r}, t)) d\mathbf{r} = \frac{1}{V} \int_V \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t)) d\mathbf{r} \tag{4.16}$$

We now take the ensemble average of both sides of this relation, but in order to have an expression involving the characteristic function, we would like to move the average signs *inside* the integral signs. As far as the Eulerian integral is concerned, we can retain the integration volume to be the same, without further ado, provided the flow is incompressible, as we have assumed from the outset here. Lagrangian integral is problematic, however. Here, the volume is deformed as the flow defining the volume evolves. The *relative* deformation is however, immaterial in the limit where $V \to \infty$, simply because the number of particles leaving the reference volume scales with the *surface* of V (Tennekes and Lumley, 1972). Consequently, we can formally consider the surface of this volume as fixed also for the Lagrangian case. In the limit $V \to \infty$ we therefore have

$$\begin{aligned}
&\lim_{V\to\infty} \frac{1}{V} \left\langle \int_V \exp(i\mathbf{k} \cdot \mathbf{U}_E(\mathbf{r}, t)) d\mathbf{r} \right\rangle = \\
&\lim_{V\to\infty} \frac{1}{V} \int_V \langle \exp(i\mathbf{k} \cdot \mathbf{u}_E(\mathbf{r}, t)) \rangle d\mathbf{r} = \\
&\lim_{V\to\infty} \frac{1}{V} \left\langle \int_V \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t)) d\mathbf{r} \right\rangle = \\
&\lim_{V\to\infty} \frac{1}{V} \int_V \langle \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t)) \rangle d\mathbf{r}
\end{aligned} \tag{4.17}$$

For homogeneous and isotropic turbulence we have $\langle \exp(i\mathbf{k} \cdot \mathbf{u}_E(\mathbf{r}, t) \rangle$ as well as $\langle \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t) \rangle$ being independent of position (that is the whole idea with homogeneity and isotropy) and we can take these quantities outside

the integration. The integrations are then trivially giving the volume, which cancels with the volume in the divisor and we and up with the relation

$$\langle \exp(i\mathbf{k} \cdot \mathbf{u}_E(\mathbf{r}, t)) \rangle = \langle \exp(i\mathbf{k} \cdot \mathbf{u}_L(\mathbf{r}(t), t)) \rangle \,. \tag{4.18}$$

When the characteristic functions for the Eulerian and the Lagrangian velocities are equal, we trivially have $\langle u_E^2 \rangle = \langle u_L^2 \rangle$ and we did right in avoiding any subscript in (4.12). This is, however, correct *only* for homogeneous and isotropic incompressible turbulence. It is essential for the argument that the integration volume is a known constant in (4.16) and (4.17). This is the case because the plasma is assumed to be incompressible, i.e. the plasma fills the same area at all times in our two dimensonal model.

The velocity field mediating the turbulent transport is *for absolute diffusion* a stationary random process. The particle displacement is of course steadily increasing in a mean square sense, and is therefore *not* a stationary random process.

Chapter 5

Numerical Model

To study the implications of larger ensembles of vortices, especially the transport properties of a random ensemble, we have implemented the model described in chapter 3 in a *C++* program. The source code is included in appendix A. This chapter describes the implementation and the routines used to simulate the system.

5.1 Assumptions and approximations

- The magnetic field $\mathbf{B}$ is assumed to be purely in the $\hat{\mathbf{z}}$ direction and all variation in the plasma is assumed to be in the plane perpendicular to $\hat{\mathbf{z}}$.
- The magnetic field is assumed to be homogeneous in the plane perpendicular to $\hat{\mathbf{z}}$.
- The fields are electrostatic, and the plasma currents are not strong enough to perturb the magnetic field in the perpendicular direction.
- The basic state of the plasma is homogeneous and isotropic. The vortex structures are introduced as perturbations in the electrostatic potential by introducing line charges.
- All dynamics are assumed to occur on time-scales much longer than the cyclotron frequencies and on length scales much longer than the Larmor radius. This allows us to only consider the guiding centre motion and view the line charges as charged magnetic field lines.

To simulate the motion of electrostatic vortices, as described in section 3.1 we will use a discretized version of the equations of motion for an ensemble of

vortices; equation (3.33) with Debye shielding and equation (3.34) without. We will solve these equations numerically using the Runge-Kutta algorithm to the 4th order.

5.1.1 Dimensions

When representing numbers on a computer, we are limited by the maximum precision, called *double precision*, allowed in most computer applications. When we work with numbers close to this limit, the errors in the representation grows. This means that we will often like to express simulated system in units which are typical for the physical system. This is done by dividing our physical quantity with a number with the same unit, and which is typical for the system.

Two natural scales present themselves in this problem, the Debye length $\lambda_D = \sqrt{\frac{\varepsilon_0 T_0}{n_0 q_0^2}}$, and the $\mathbf{E} \times \mathbf{B}$-velocity $\mathbf{U}_0 = \frac{\mathbf{E}_0 \times \mathbf{B}_0}{B_0^2}$ where $E_0 = \frac{\gamma_0}{2\pi\varepsilon_0\lambda_D}$. We then retrieve a characteristic line charge density $\gamma_0 = q_0/\lambda_D$ for two dimensional plasmas. Here subscript 0 refer to an arbitrary reference plasma. By using these natural scales, a natural time-scale $t_0 = \frac{\lambda_D}{U_0}$ can be constructed. By inserting the expressions for the Debye length and the $E \times B$ velocity we get

$$t_0 = \frac{2\pi\varepsilon_0^2 B_0 T_0}{e^2 n_0 \gamma_0}.$$

All times will be expressed in this scale, which corresponds to the time it takes for a particle moving with velocity U_0 to travel one Debye length, given E_0, B_0, T_0, n_0 of an arbitrary plasma.

All quantities given in this, or later chapters, are, unless otherwise stated, normalized with the corresponding reference values.

5.2 Equations of motion

Since the vortices move like massless particles, their velocity is, at all times, the instantaneous $\mathbf{E} \times \mathbf{B}$-velocity generated by the ensemble of all vortices. The whole motion of one vortex in the field of all others is then described by the equation

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{U}, \tag{5.1}$$

with the two component equations

$$u_i = \frac{\mathrm{d}x_i}{\mathrm{d}t} \quad \text{In the x-direction} \tag{5.2}$$

$$v_i = \frac{\mathrm{d}y_i}{\mathrm{d}t} \quad \text{In the y-direction} \tag{5.3}$$

where, as was described in section 3.1, the velocity $\mathbf{U} = (u, v)$ is the instantaneous $\mathbf{E} \times \mathbf{B}$-velocity, with direction $\mathbf{E} \times \mathbf{B}$ and magnitude E/B. $\mathbf{E}$ is the instantaneous electric field from the vortex $j \neq i$ evaluated in the position of vortex i, and $\mathbf{B} = B\hat{\mathbf{z}}$, where $\hat{\mathbf{z}}$ is the unit vector in the z direction, is the homogeneous magnetic field in which the vortices are moving.

By insertion of the expression for the velocity in (5.2) and (5.3) the equations of motion becomes

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = \frac{E_y B}{B^2}, \tag{5.4}$$

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = -\frac{E_x B}{B^2}. \tag{5.5}$$

5.2.1 Unshielded vortices

If the vortices are unshielded, the electrostatic potential of one vortex at position $\mathbf{r}$ due to another in position $\mathbf{r}_\alpha$ is $\phi = \gamma_\alpha \ln(|\mathbf{r} - \mathbf{r}_\alpha|)$, as described in section 3.1. This gives the electric field on one vortex in $\mathbf{r}$ from another in $\mathbf{r}_\alpha$ as

$$\mathbf{E} = \gamma_\alpha \frac{\mathbf{r} - \mathbf{r}_\alpha}{(|\mathbf{r} - \mathbf{r}_\alpha|)^2}. \tag{5.6}$$

5.2.2 Shielded vortices

If, on the other hand, the vortices are shielded, the potential of one vortex at position $\mathbf{r}$ due to another in position $\mathbf{r}_\alpha$ is modified to (equation (3.31)) $\phi = \gamma_\alpha K_0(|\mathbf{r} - \mathbf{r}_\alpha|)$ where K_0 is the modified Bessel function of the second kind of order zero. This gives the electric field on one vortex in $\mathbf{r}$ from another in $\mathbf{r}_\alpha$ as

$$\mathbf{E} = \gamma_\alpha \frac{\mathbf{r} - \mathbf{r}_\alpha}{|\mathbf{r} - \mathbf{r}_\alpha|} K_1(|\mathbf{r} - \mathbf{r}_\alpha|). \tag{5.7}$$

Where K_1 arises from the differentiation of K_0 and is the modified Bessel function of the second kind of order one.

5.2.3 Modified Bessel functions

The modified Bessel functions of the second kind (sometimes known as modified Bessel functions of the third kind) of order n for a real argument x can be found from the recursion formula

$$K_{n+1}(x) = K_{n-1}(x) + \frac{2n}{x}K_n(x), \tag{5.8}$$

starting from $K_0(x)$ and $K_1(x)$. For the implementation of Debye shielded vortex potentials we need to calculate $K_1(x)$. We calculate $K_1(x)$ from two different polynomial approximations, one for $0 < x \leq 2$ ($b1$), which has truncation error $|\epsilon < 8 \times 10^{-9}|$, and one for $2 \leq x < \infty$ ($b2$) with truncation error $|\epsilon| < 2.2 \times 10^{-7}$. Using $b1$ requires the calculation of the modified Bessel function of the first kind of order one I_1 from it's polynomial approximation valid for $-3.75 \leq x \leq 3.75$ with truncation error $|\epsilon| < 8 \times 10^{-9}$ (Abramowitz and Stegun, 1965: Handbook of Mathematical Functions).

All modified Bessel functions needed for this study were implemented in the routine called *tbessk.cpp*. This contains the functions necessary to calculate $K_n(x)$ and was originally written by Jean-Pierre Moreau[1].

5.3 4th order Runge-Kutta algorithm

Integration of the equations of motion are performed with the 4th order Runge-Kutta algorithm with variable time step, which, given an initial value problem

$$\dot{y} = f(t, y), \quad y(t_0) = y_0 \tag{5.9}$$

is as follows:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{5.10}$$

$$t_{n+1} = t_n + h \tag{5.11}$$

where h is the iteration step length, y_{n+1} is the RK4 approximation of $y(t_{n+1})$ and

$$k_1 = hf(t_n, y_n) \tag{5.12}$$

$$k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1) \tag{5.13}$$

$$k_3 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2) \tag{5.14}$$

$$k_4 = hf(t_n + h, y_n + k_3) \tag{5.15}$$

[1] **Source:** *http://jean-pierre.moreau.pagesperso-orange.fr/c_bessel.html*

The next value (y_{n+1}) is given by the previous value (y_n) and the weighted average of four increments where each increment is the product of the step length h and a slope estimated by the function f. From equations (5.12)-(5.15) we have

- k_1 is the increment based on the slope at the beginning of the interval y_n, and corresponds to Euler's method
- k_2 is the increment based on the slope at the midpoint of the interval, using $y_n + \frac{1}{2}k_1$
- k_3 is another increment based on the slope at the midpoint, but now using k_2
- k_4 the increment using the slope at the end of the interval $y_n + k_3$

The weighted average is based on Simpson's rule. RK4 is a fourth order method, meaning that the stepwise error is on the order of h^5 ($\mathcal{O}(h^5)$) and the accumulated error is on the order of h^4 ($\mathcal{O}(h^4)$).

Alternative integration schemes, like Euler's method, Leapfrog and 2nd order Runge-Kutta algorithm leads to accuracies $\mathcal{O}(h)$, $\mathcal{O}(h^2)$ and $\mathcal{O}(h^2)$ respectively. The Euler and Leapfrog methods are the fastest, both Runge-Kutta methods perform more calculations, and the Leapfrog method would be preferred over 2nd order Runge-Kutta if 2nd order accuracy was sufficient. However for the vortex simulations carried out here the extra accuracy of the RK4 scheme is of sufficiently great value that the relative slowdown of the code is an acceptable sacrifice.

5.4 Discretized equations of motion

Discretizing the equations of motion, (5.2) and (5.3), results in the set of two coupled equations for vortex i on the form of the RK4-algorithm given above:

$$x_{i,n+1} = x_{i,n} + \frac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}) \quad \text{In the } x\text{-direction} \tag{5.16}$$

$$y_{i,n+1} = y_{i,n} + \frac{1}{6}(k_{1y} + 2k_{2y} + 2k_{3y} + k_{4y}) \quad \text{In the } y\text{-direction.} \tag{5.17}$$

$k_{1a}, k_{2a}, k_{3a}, k_{4a}$ in the a-direction are estimates of the position at $t + \Delta t$ as described above. Given the initial conditions

$$x_i(t = 0) = x_0 \tag{5.18}$$

$$y_i(t = 0) = y_0 \tag{5.19}$$

and estimates of the slopes

$$f_{x_i}(t, x_i) = U_x = \frac{\sum_{j \neq i} E_{y,j} B}{B^2} \tag{5.20}$$

$$f_{y_i}(t, y_i) = U_y = -\frac{\sum_{j \neq i} E_{x,j} B}{B^2} \tag{5.21}$$

where, again, $E_{y,j}$ is the electric field from vortex j evaluated at the position of vortex i. By this we have a closed system of discretized equations, which can be iterated to find the positions $(x_{i,n+1}, y_{i,n+1})$.

5.4.1 Initial conditions

We are most interested in studying random ensembles of many vortices. The random initial conditions are generated by using a random number generator. For every particle we generate two random numbers to give a uniform distribution in a specified area of the plane. We also generate a random number from some distribution to give the vortex strength. The initial velocity of particle i is then calculated from the positions of all the vortices $i \neq j$ and the vortex strength

The random numbers needed to generate the initial conditions for the simulations are generated from the standard C++ random number generator (RNG) *drand48()*, which is seeded by the function *srand48(seed)*. Some simulations were run with constant $seed = 1$. This results in the same sequence of pseudo random numbers being generated in every simulation. In other cases we wanted to have a varying seed for every simulation; in these cases we seeded the RNG with the computer clock using the C++ function *time(NULL)*.

5.5 Time step

The system consisting of an ensemble of line charges is susceptible to errors if the time resolution is inadequate. If this is the case the system might not be simulated correctly, which can result in unphysical behaviour, so care must be taken to assure that the time step used in the simulation is adequately small. For the cases covered in this thesis this problem is of particular concern for the initial time steps of a simulation of the diffusion of a cloud and for the entire simulation if we have imposed periodic boundary conditions. In the latter case the overall density of line charges will not decrease since no vortex is allowed to exit the reference cell.

For every time step of the simulation, we find the shortest distance separating two vortices l_{min} and the highest calculated velocity experienced by a vortex U_{max}. We use these to calculate the necessary time step through the following reasoning: If the shortest separation distance is the radius in a circular orbit, and the vortex following that orbit will travel with velocity U_{max}, the time step Δt should be such that the vortex will use k time steps to travel one orbit if the velocity remained unchanged. The Nyquist sampling condition tells us that we have to have at least $k = 4$ samples, i.e. the vortex has to use 4 time steps to complete one orbit, to be able to simulate the orbit correctly. To be conservative, we have chosen to use $k = 10$ samples as the condition for the time step Δt; the time step is then calculated from the condition $U_{max}\Delta t = \pi l_{min}/k$, which gives

$$\Delta t = \frac{\pi l_{min}}{k U_{max}}$$

5.5.1 Calculation of the Hamiltonian

The Hamiltonian of the simulated unshielded vortex systems has been calculated using equation (3.18), ignoring a numerical constant, and should be constant in time for a given realization. A drift in the value of the Hamiltonian would indicate numerical errors, possibly arising from the time resolution not being adequate.

In Figure 5.1 we show the Hamiltonian as function of time for one realization of a homogeneous system of vortices (see sections 5.8). The mean value was $\langle H(t)\rangle = -0.0258$, the maximum value was $H_{max} = -0.0236$ while the minimum value was $H_{min} = -0.0293$. The ratio $H_{max}/H_{min} = 0.8035$. There are rather large fluctuations in the value of the Hamiltonian, but there is no overall trend with time. That the value is not perfectly conserved is not unexpected in a numerical simulation. Not as well that the initial value $H(t = 0) = -0.0251$ is very close to the mean value.

5.6 Test particles

For the sake of the Lagrangian statistical analysis we have included test particles that are transported as passive scalars in the velocity field generated by the vortex distribution, see Section 4.2.1. The test particles are transported using the velocity field from all the normal vortices, and then moved in the same way. They do not, however contribute to the velocity field for other vortices. At every sampling, we write the positions and velocity components

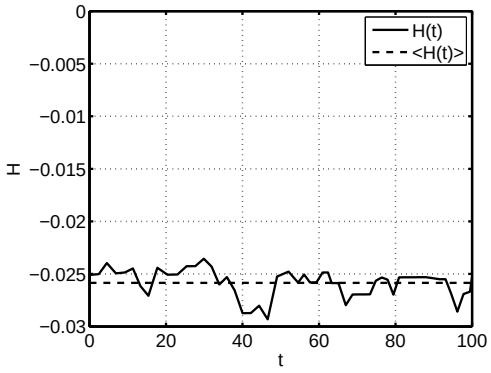


Figure 5.1: *The Hamiltonian as function of time for one realization of our vortex system.*

of these to a file. These become the raw data for the Lagrangian statistical analysis.

5.7 Eulerian points

To do Eulerian statistical analysis we chose a number of random points in plane, inside the reference cell, and at every sampling we calculate the velocity field which would have been experienced by a test particle at that position at that time. We write this to a file and use this as the raw data for the Eulerian statistical analysis.

5.8 Homogeneous systems

A homogeneous system, filling all of space, is a common approximation for large systems in equilibrium far from any boundaries. We are interested in studying the transport properties of large homogeneous ensembles of vortices. This introduces the obvious problem of calculating infinitely many interactions and propagating infinitely many equations.

5.8.1 Periodic boundary conditions

By introducing periodic boundary conditions we are able to avoid some of the obvious challenges posed above.

Periodic boundary conditions are a technique that changes the topology of the simulated space. If a vortex in the simulation leaves a predetermined

area of the plane through one edge, hereafter referred to as the unit cell, will re-enter the *unit cell* from the opposite side and with unchanged velocity. The shape of the unit cell has to be chosen in such a way that it can provide a complete tiling of the space. We have for simplicity chosen a rectangular cell, but all simulations are run with quadratic cells with side length L and with the origin in the centre of the cell.

The space around the unit cell is then tiled with *image cells* which are identical to the unit cell, and of which there are, in principle, infinitely many. This is possible due to the assumption of homogeneity; i.e. since all areas of the plane have the same statistical properties, all areas can be approximated as identical. We refer to the "layer" of 8 image cells right outside the unit cell as layer 1. The next "layer" of 16 cells as layer 2, and generalizing the kth layer will have $k^2 - (k-1)^2$ image cells. The number of layers k has in principle no upper limit, i.e. $k = 0, 1, 2, 3, ...$ The centre of each image cell is in the position (mL, nL) where $m, n = -k, -k+1, ..., -1, 0, 1, .., k-1, k$, with $(0, 0)$ being the unit cell. Figure 5.2 shows the unit cell and the first layer of mirror cells to illustrate the labelling system for the cells.

The system in the unit cell is the propagated, while taking account of interactions from the image cells with the unit cell vortices. By this, the problem of having to propagate infinitely many vortices is solved, but there still remains the problem of infinitely many interactions, as we in general have not limited ourselves to a maximum number of image-cells. Electric fields are long range, so we have chosen to let every vortex in the unit cell interact with all other vortices in the unit cell and every vortex in a "layers" of image cells. Thus the simulation becomes:

1. A number N of vortices are initialized in the unit cell

2. The velocity of vortex $i = 1, 2..., N$ at position $\mathbf{r}_i$ is calculated as

$$U_x = \left(\sum_{j=1, j\neq i}^{N} E_{y,j}(|\mathbf{r}_i - \mathbf{r}_j|) \right. \tag{5.22}$$
$$+ \sum_{m=-a}^{-1} \sum_{n=-a}^{-1} \sum_{j=1}^{N} E_{y,j}(|\mathbf{r}_i - ((mL\hat{\mathbf{x}} + nL\hat{\mathbf{y}}) + \mathbf{r}_j)|)$$
$$\left. + \sum_{m=1}^{a} \sum_{n=1}^{a} \sum_{j=1}^{N} E_{y,j}(|\mathbf{r}_i - ((mL\hat{\mathbf{x}} + nL\hat{\mathbf{y}}) + \mathbf{r}_j)|) \right) \frac{1}{B}$$

   where $\mathbf{r}_j$ is the position of vortex j. The expression is similar in the y-direction.

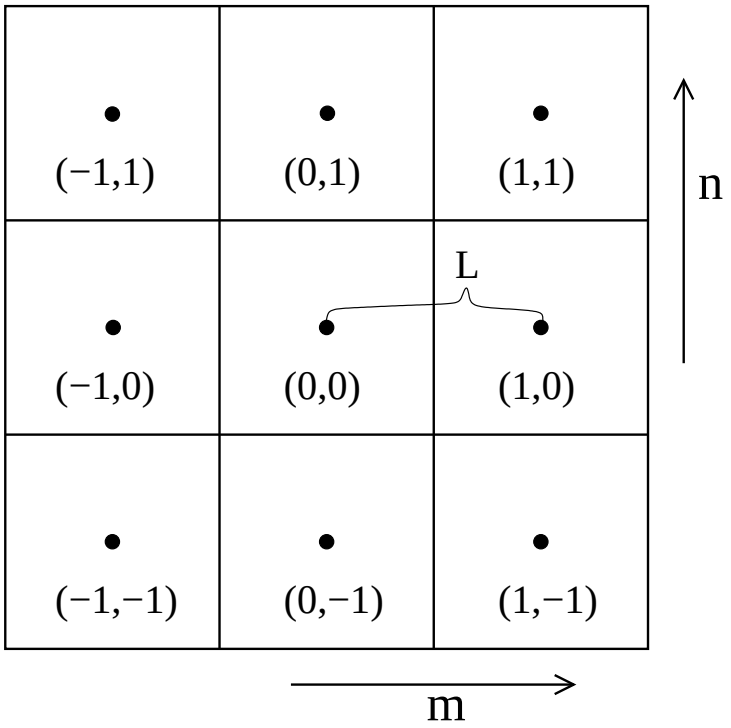


Figure 5.2: *Illustration of the unit cell and the first "layer" of image cells.*

3. The positions of the N vortices in the unit cell are propagated one time step Δt.

4. The process is repeated from point 2.

Chapter 6

Data analysis

For the purpose of analysing the simulations, we made datasets from each run. The main variables contained in these datasets was the position of each vortex, the positions and velocities of the test vortices and the velocity at each of the Eulerian points at every sampling time. We have used these to study the time evolution of the vortex systems from several different initial conditions. The main focus of this study is the characterisation of the transport properties of a random vortex ensemble, both infinite and localized.

6.1 Density histograms

A basic characteristic of a localized system is it's spatial distribution as a function of time. To study this we have made density histograms, which show the particle density in a particular area of space at the sampling time

The density histograms are made with a two dimensional histogram routine resulting in plots as shown in Figure 7.1. Temporal evolutions can be tracked through comparisons of diagrams taken at selected times.

6.2 Ensemble mean values

The raw data are used to directly calculate the first order moment $\langle |r| \rangle$ and the second order $\langle r^2 \rangle$ moments, respectively, of the spatial distribution at every sampling time. These characterize the average time evolution of the localized ensembles. The second order moment is the variance of the spatial distribution, given that $\langle r \rangle = 0$, which is the case here, while $\langle |r| \rangle$ is also a measure of the spread of the distribution, but without obvious physical meaning.

If we take the square root of the second order moment, i.e. $\sqrt{\langle r^2 \rangle}$, this is the root mean square expansion of the ensemble and is also the standard deviation of the spatial distribution. This has the same physical dimensions as $\langle |r| \rangle$, but the two will generally not give the same results, due to subtle differences in what they measure. It can be generally shown that $\sqrt{\langle r^2 \rangle} \geq \langle |r| \rangle$ (See below on the Schwarz inequality.).

6.2.1 Schwarz' inequality

We have the general relation called Schwarz' inequality

$$\int f^2(r)dr \cdot \int g^2(r)dr \geq \left[\int f(r)g(r)dr \right]^2 . \tag{6.1}$$

We use this inequality to compare $\sqrt{\langle r^2 \rangle}$ and $\langle |r| \rangle$. For cylindrically symmetric distributions we have $\langle |r| \rangle \equiv \int_0^\infty rG(r)dr / \int_0^\infty G(r)dr$ and $\langle r^2 \rangle \equiv \int_0^\infty r^2 G(r)dr / \int_0^\infty G(r)dr$, where we introduced $\int_0^\infty G(r)dr$ for normalization. We define $g = r\sqrt{G}$ and $f = \sqrt{G}$ and find

$$\frac{\int_0^\infty G(r)dr \cdot \int_0^\infty r^2 G(r)dr}{\left(\int_0^\infty G(r)dr \right)^2} \geq \frac{\left[\int_0^\infty rG(r)dr \right]^2}{\left(\int_0^\infty G(r)dr \right)^2}. \tag{6.2}$$

Simplifying the above expression results in

$$\left\langle r^2 \right\rangle \geq (\langle |r| \rangle)^2, \tag{6.3}$$

which is generally valid for cylindrically symmetric distributions.

The same calculation becomes surprisingly complicated even when the distribution is elliptical instead of circular. We could not show this generally even using the symbolic integration in *Mathematica*. The same result was however proven through numerical integration for one specific elliptical distribution, and it is expected to hold in general.

6.3 Interpolation

Since the simulation is run using variable time step, the data are not evenly sampled in time, however for various reasons, we would like to have the data evenly sampled in time. To get this we can interpolate the original dataset into one which is evenly spaced. There are many interpolation methods, e.g. linear, polynomial, spline, and in choosing one method we have gone for the simplest, namely linear interpolation.

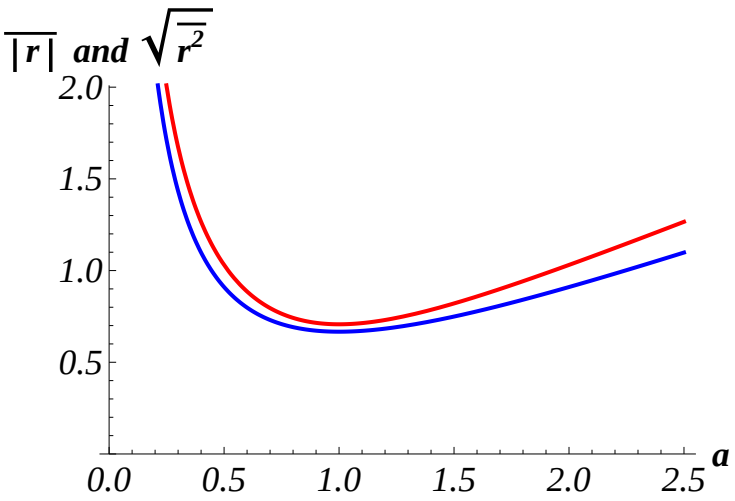


Figure 6.1: $\sqrt{\langle r^2 \rangle}$ *(red) and* $\langle |r| \rangle$ *(blue) for an elliptical distribution as function of the eccentricity a of the distribution.*

Generally linear interpolation is done by taking two data points (t_a, x_a) and (t_b, x_b) and finding the interpolant

$$x = x_a + (x_b - x_a)\frac{(t - t_a)}{(t_b - t_a)} \quad \text{at the point}(t, x) \tag{6.4}$$

This method has an error on the order $\mathcal{O}(\Delta t^2)$, which is quite large. In our application of this we are not interested in increasing the resolution of our data sets to a higher number of points, but only to resample the data to a regular grid. Due to this, and the fact that the time step is generally small, $\Delta t < 5 \cdot 10^{-3} t_0$, linear interpolation will be good enough as long as the analysis involves frequencies that are larger than $1/\Delta t$.

We have used the standard interpolation function i MATLAB (*interp1*) with the option *linear*, to do linear interpolation.

6.4 Correlation functions

A correlation function is a measure of the correlation between two random variables separated in time (or space). It is a measure of how closely a data set at time t_1 resembles another data set at a different time t_2. These data sets can be the same or different, and these two cases are called autocorrelation functions and cross correlation functions.

The cross correlation function gives the correlation between two different time series, and can be considered a measure of how well one measured

quantity can be used to predict the development of another; it is important to remember that we do not necessarily know which quantity is cause and which is effect. The cross correlation function is defined as

$$C(t_1, t_2) = \frac{\langle X(t_1)Y(t_2)\rangle}{\sqrt{\langle X^2\rangle}\langle Y^2\rangle} \tag{6.5}$$

where $X(t), Y(t)$ are two different data sets; $\langle X(t_1)Y(t_2)\rangle$ is the covariance of X and Y.

The autocorrelation function is the cross correlation between a time series and itself, and gives the correlation between two points from the same dataset at different points in time. It can be considered as a measure of how well measurements of one quantity at earlier times can be used to predict the development of the same quantity at later times. The autocorrelation function is defined similarly

$$C(t_1, t_2) = \frac{\langle X(t_1)X(t_2)\rangle}{\langle X^2\rangle}. \tag{6.6}$$

If the random process measured is time stationary, we can simplify the above expressions by introducing τ which denotes the time difference between t_1 and t_2, i.e. $\tau = t_2 - t_1$. We are mainly interested in calculating correlation functions for the simulated infinite systems and assuming a time stationary process seems reasonable in that case, since an infinite system in equilibrium is per definition time stationary. The new expression for the cross correlation becomes

$$C_{XY}(\tau) = \frac{\langle X(t)Y(t+\tau)\rangle}{\sqrt{\langle X^2\rangle \langle Y^2\rangle}}. \tag{6.7}$$

The autocorrelation is then similarly

$$C_{XX}(\tau) = \frac{\langle X(t+\tau)X(t)\rangle}{\langle X^2\rangle}. \tag{6.8}$$

The discretized expression for the cross correlation functions is

$$C_{XY} = \frac{\frac{1}{n-\tau}\sum_{1=1}^{n-\tau} X(i)Y(i+\tau)}{\sqrt{\langle X^2\rangle \langle Y^2\rangle}} \tag{6.9}$$

and equivalently for the autocorrelation function.

The expressions above give the correlation functions between two time series. Since we have several datasets, it will be convenient to combine the results into one correlation function. The correct way to find this for m

datasets is given below, where the covariance between the two variables X and Y and the mean squares of X and Y are found separately.

$$C_{XY}(\tau) = \frac{\frac{1}{m}\sum_{j=1}^{m}\left(\frac{1}{n-\tau}\sum_{1=1}^{n-\tau} X(i)Y(i+\tau)\right)_j}{\sqrt{\frac{1}{m^2}\sum_{j=1}^{m}\langle X^2\rangle_j \sum_{j=1}^{m}\langle Y^2\rangle_j}} \tag{6.10}$$

Physically, the correlation function quantifies how the processes which generate a signal are related, and the ability to use one signal to predict or explain the variation in in the same or another signal. For the autocorrelation function this relation is always greatest for $\tau = 0$. The cross correlation function can have it's maximum value for any τ depending on how the two processes are related. The correlation functions contain physical information on e.g. time scales of a process or the delay between two related processes.

In the current study it will be most interesting to investigate how the velocity of a test particle ($\mathbf{u}(\mathbf{r}(t), t)$) is correlated with itself, and the difference between the velocity autocorrelation function of test particles and velocity autocorrelation functions of Eulerian points (autocorrelation of $\mathbf{u}(\mathbf{r}, t)$). These two different autocorrelation functions are called, respectively, the Lagrangian velocity autocorrelation function and the Eulerian velocity autocorrelation function, and their theoretical differences are discussed in detail in Chapter 4

For a time stationary process the autocorrelation function is symmetric around $\tau = 0$. If we calculate the Fourier transform of the autocorrelation function this symmetry results in a Fourier transform which is real and positive for all frequencies. This is the power spectrum of this variable. The power spectrum gives information about the frequencies of the energy containing processes in the system. From Section 4.2.1 we remember that the power at frequency $\omega = 0$ gives the Lagrangian integral correlation time τ_L.

Since we will have finite length time series of data, the autocorrelation functions for high time lag τ will be quite noisy. To avoid having this introducing spurious fluctuations in the power spectra, we will only use the first 2/3s of the autocorrelation functions when calculation the power spectra.

A finite number of realizations, such as we have here, can only provide an estimate of the average, not the true values. To quantify the error done by having a finite number of realizations can be found through the following:

Consider a random variable x and assume that you have found an estimate $\frac{1}{N}\sum_{j=1}^{N} x_j$ for the average $\mu \equiv \langle x\rangle$. The error e on the estimate is given as

$$e \equiv \left\langle \left(\langle x\rangle - \frac{1}{N}\sum_{j=1}^{N} x_j \right)^2 \right\rangle. \tag{6.11}$$

This results in

$$e = \langle x \rangle^2 + \frac{1}{N^2} \left\langle \left(\sum_{j=1}^{N} x_j \right)^2 \right\rangle - 2 \left\langle \langle x \rangle \frac{1}{N} \sum_{j=1}^{N} x_j \right\rangle .$$

The last term can be written as $-2 \langle x \rangle^2$, while the second term can be written as

$$\frac{1}{N^2} \sum_{k=1}^{N} \sum_{j=1}^{N} \langle x_j x_k \rangle .$$

This double sum contains N terms where $j = k$, all of which results in $\langle x_j^2 \rangle = \langle x^2 \rangle$, and $N(N-1)$ terms where $j \neq k$. The latter can be written as $\langle x_j x_k \rangle = \langle x_j \rangle \langle x_k \rangle = \langle x \rangle^2$. Combining these terms again, our error e is

$$e = \langle x \rangle^2 + \frac{1}{N^2} (N \langle x^2 \rangle + N(N-1) \langle x \rangle^2) - 2 \langle x \rangle^2 .$$

Simplifying results in

$$e = \frac{1}{N} (\langle x^2 \rangle - \langle x \rangle^2).$$

For $N \to \infty$ we have $e \to 0$ as expected in such a way that the root-mean-square error decreases as $1/\sqrt{N}$. The discussion can be extended to other, also higher order, averages and correlations, but the expressions become lengthy.

6.5 Probability density functions

The measured time series can be difficult to interpret in their raw state. To get an overall impression of the data we have estimated the probability density function (PDF) of the data for the Lagrangian and Eulerian velocity statistics. To do this we grouped the data into bins and counted the number of observations in each bin, just as one would for making a histogram. The result was the normalized to ensure that the area under the graph was equal to one.

Chapter 7

Results

In this chapter we present the results from the numerical experiments conducted. All results are obtained from simulations done with the code described in Chapter 5.

First we present results concerning the time evolution of a localized random ensemble of N vortices with net vorticity density $\frac{1}{N}\sum_i \gamma_i$. We present the results for several different initial conditions.

Next we will study a generalization of the four vortex collisions from section 3.2.3. We generalize this to four ensembles with $N = 100$ individual vortices in each, and we will classify the behaviour in terms of the impact parameter b.

Last we present the results from simulations of homogeneous systems in terms of a turbulent velocity field, as discussed in Chapter 4, by using the boundary conditions presented in section 5.8.

7.1 Time evolution of a localized random ensemble

The basic behaviour of such systems depends on the sign and distribution of the vortex strengths. We adopt the notation $\chi \sim D(a, b)$ where the variable χ is a stochastic variable with distribution $D(a, b)$. The relevant cases are

1. All vortices have the same sign
    - All vortices have the same sign and strength, set to 1.
    - The vortex strengths are distributed $\gamma \sim U(0, 1)$[1]. Here $U(a, b)$

[1]In all these cases $P(\gamma = 0) = 0$ since vortices with $\gamma = 0$ will be completely inert. Simulating these will be a waste of computational resources.

represent the uniform distribution on the interval (a, b)

2. The vortex strengths are randomly distributed with mean 0 so that the total vorticity is $\frac{1}{N}\sum_i \gamma_i \approx 0$.

    - The strengths are distributed so that $P(\gamma = 1) = P(\gamma = -1) = 1/2$.
    - The strengths are distributed $\gamma \sim U(-1, 1)^1$.
    - The strengths are distributed $\gamma \sim N(0, 1/2)^1$. Here $N(a, b)$ represents the Normal distribution with $\mu = a$ and $\sigma = b$.

The cases have been simulated using the code described in Chapter 5, and the simulations have been done for both unshielded and shielded vortices as well as for both circular and elliptical spatial distributions. The circular distributions are exact solutions of the equations (3.6) and (3.7), while the elliptical distributions are not. Density histograms of these simulations are shown in Figures 7.1-7.4). The cases shown here are illustrative: Circular and elliptical distribution with $P(\gamma) = 1$; circular and elliptical distribution with $P(\gamma = 1) = P(\gamma = -1) = 1/2$. The illustrations are for unshielded vortex potentials.

The basic behaviour is, as previously stated, strongly dependent on the sign distribution of the systems and on the presence of shielding. When all vortices have the same sign, the dominating behaviour is a large scale rotation of the entire system, similar to the two vortex dynamics for same sign vortices. There is very little expansion of the system in these cases, especially for the circular distributions.

The elliptical distributions are stretched by the rotation, and the outer parts of the ellipse are thrown away from the central rotating core, which rapidly becomes circular.

When shielding effects have been included, and with spatial distribution significantly larger than the Debye length, the result is no large scale coherent motion, only small scale fluctuating interactions.

When the vortices have different polarity and the net vorticity density is approximately zero, there is no net rotation of the system, instead the small scale interactions between nearby vortices dominate, and the pairing up of two vortices of opposite polarity results in a large of expansion of these clouds. The cloud quickly loses it's character. This general behaviour is valid both for circular, and elliptic clouds and for both shielded and unshielded vortices. The result here is visually similar to the turbulent diffusion illustrated in Figure 4.1. The distortion of the cloud of vortices is much larger

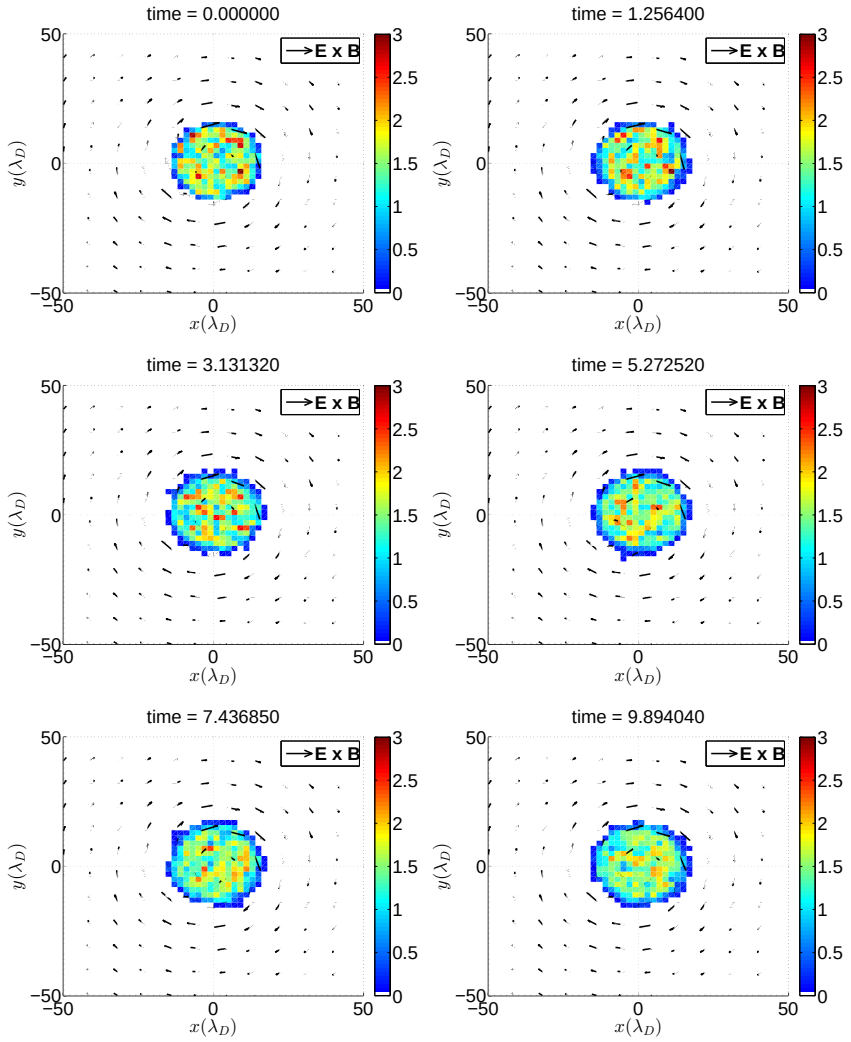


Figure 7.1: *Histograms of the time evolution of a circular cloud of line charges, all with line charge density* $\gamma = 1$.

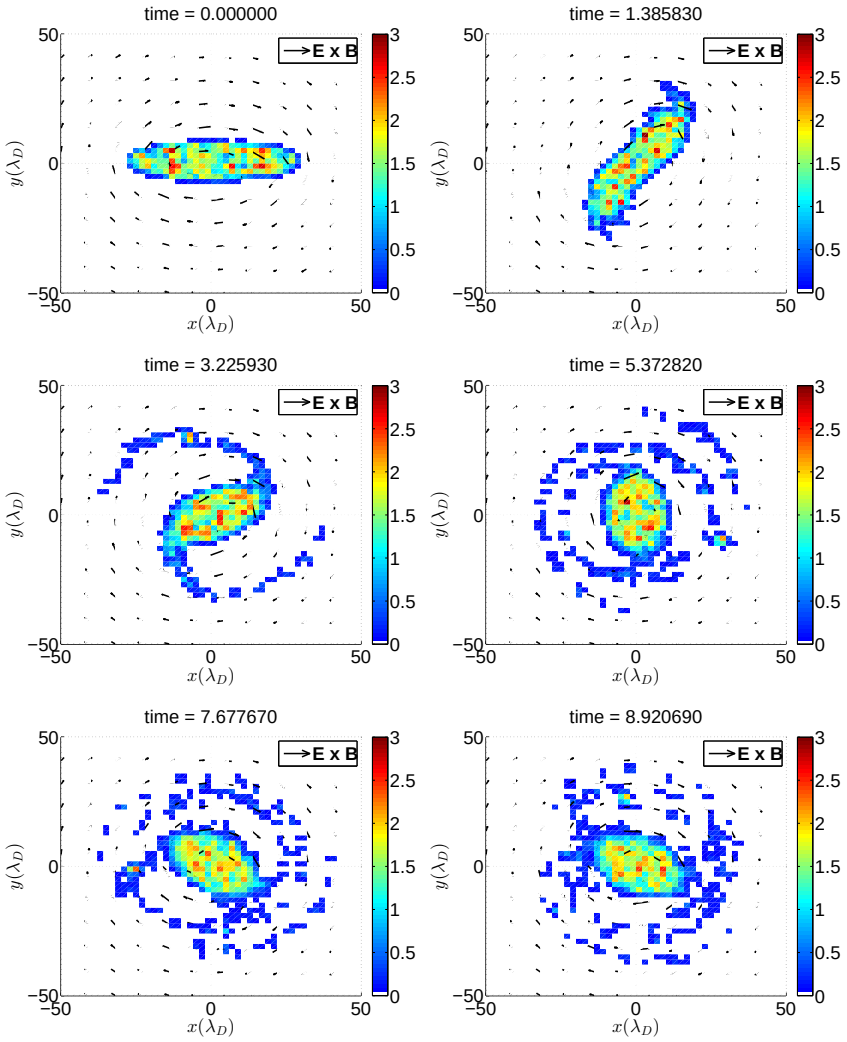


Figure 7.2: *Histograms of the time evolution of an elliptic cloud of line charges, all with line charge density* $\gamma = 1$.

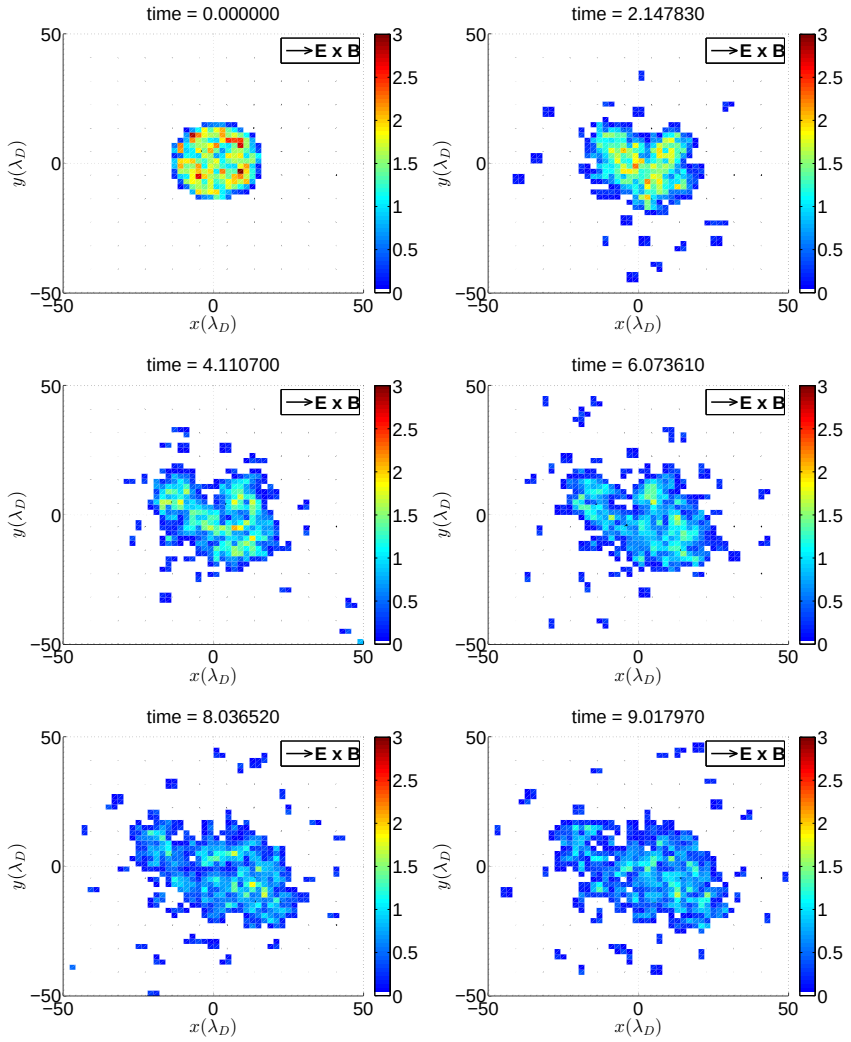


Figure 7.3: *Histograms of the time evolution of a circular cloud of line charges, all with line charge density randomly distributed to* $\gamma = \pm 1$.

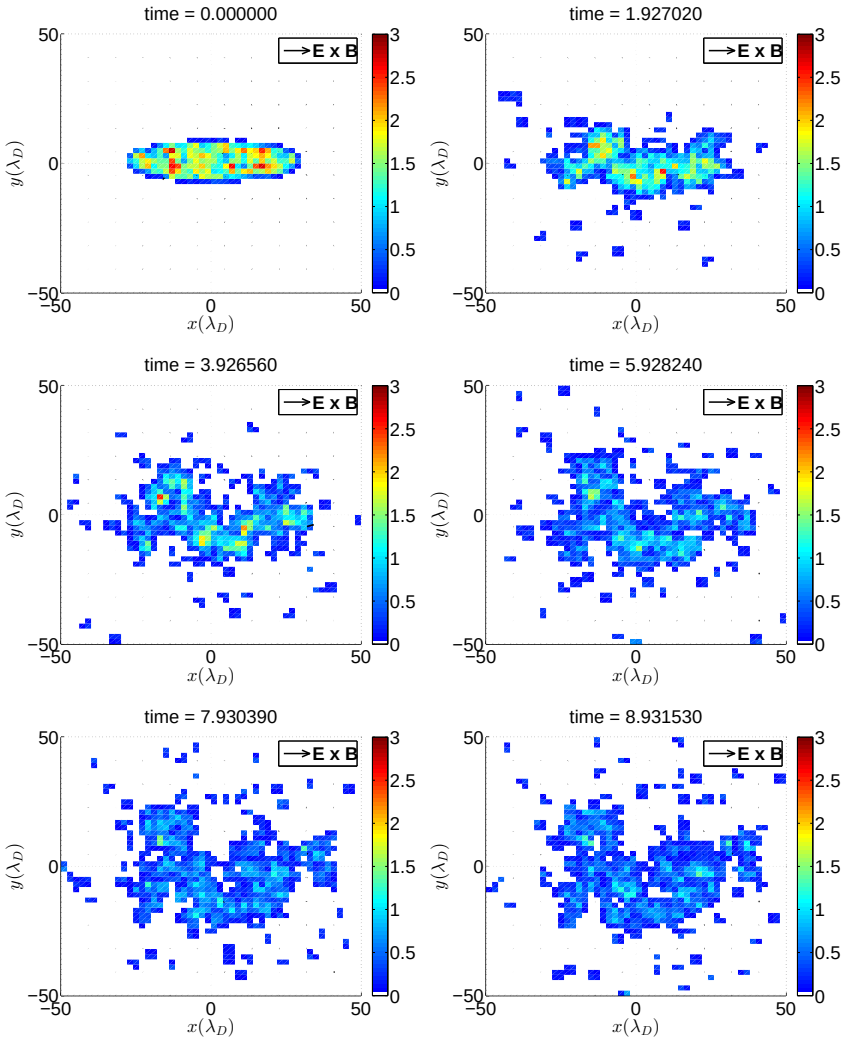


Figure 7.4: *Histograms of the time evolution of an elliptic cloud of line charges, all with line charge density randomly distributed to* $\gamma = \pm 1$.

Table 7.1: *The value of the Hamiltonian for different initial conditions of a localized cloud. The first three rows list the cases with vorticity density close to zero. The two last rows are the cases where all vortices have the same polarity.*

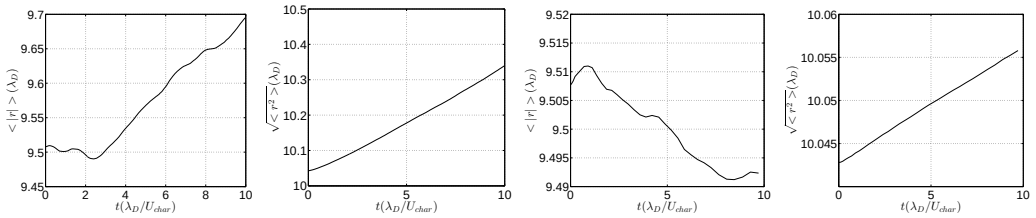
Spatial distribution	Vorticity distribution	Hamiltonian H
Circular	$P(\gamma = 1) = P(\gamma = -1) = 1/2$	$H = -1100.6$
Elliptical	$P(\gamma = 1) = P(\gamma = -1) = 1/2$	$H = 1561.3$
Circular	$\gamma \sim U(-1, 1)$	$H = 318.43$
Elliptical	$\gamma \sim U(-1, 1)$	$H = 478.55$
Circular	$\gamma \sim N(0, 1/2)$	$H = -375.73$
Elliptical	$\gamma \sim N(0, 1/2)$	$H = -242.55$
Circular	$P(\gamma = 1) = 1$	$H = -2401472$
Elliptical	$P(\gamma = 1) = 1$	$H = -2620156$
Circular	$\gamma \sim U(0, 1)$	$H = -595847.2$
Elliptical	$\gamma \sim U(0, 1)$	$H = -648655.9$

than expected for simple molecular diffusion, but for large times we expect the result to be the same, as discussed in Chapter 4.

The vortices have no kinetic energy, but there is still electrostatic potential energy in the system (see section 3.2.5). The Hamiltonian has been calculated, ignoring a numerical constant, for the localized cloud systems, allowing us to compare the Hamiltonian between cases.

We see that the Hamiltonian is more than 3 orders of magnitude larger when the vortices have the same polarity, compared to when the polarity is random. The first represents a system with high local order in the sense that for an arbitrary reference vortex, a nearby vortex will have the same polarity. This can be contrasted with a system with large negative Hamiltonian, where a vortex nearby to our reference vortex will have the opposite polarity, see section 3.5

Figures 7.5, 7.6, 7.7 and 7.8 shows the ensemble first moment (a,c) and second moment (b,d) as function of time for the simulations where all line vortices have the same polarity (1 in the numbered list). In each figure (a) and (b) shows the unshielded result while (c) and (d) show the shielded results. We see that $\langle |r| \rangle < \sqrt{\langle r^2 \rangle}$ for the ensembles, and that their functional dependences on t are not the same, even though they have the same unit. The second moment generally show a slight increase over time, corresponding to a slight expansion of the ensembles.



(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

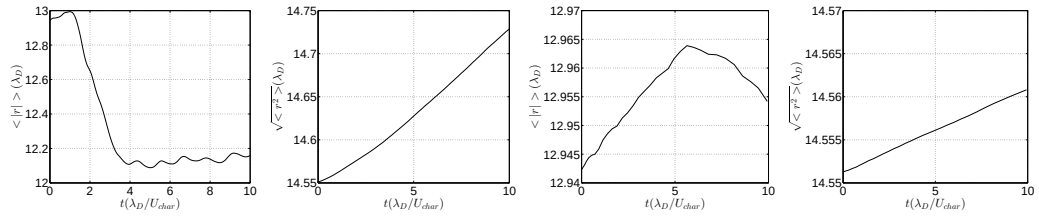
Figure 7.5: *Mean expansion (a,c) and root mean square expansion (b,d) of a circular cloud of line charges, with and without shielding, all with line charge density* $\gamma = 1$

For the circular cases (7.5 and 7.7) $\sqrt{\langle r^2 \rangle} \propto t$ for all but the shortest time scales. This corresponds to ballistic expansion of the ensemble. We would expect that for larger t, the expansion would tend to become diffusive. The first moment is much more irregular.

The elliptic cases show the same tendency for the case where $\gamma_j = 1$ with ballistic expansion, but the case with uniform distribution is highly irregular, and shows no real total expansion.

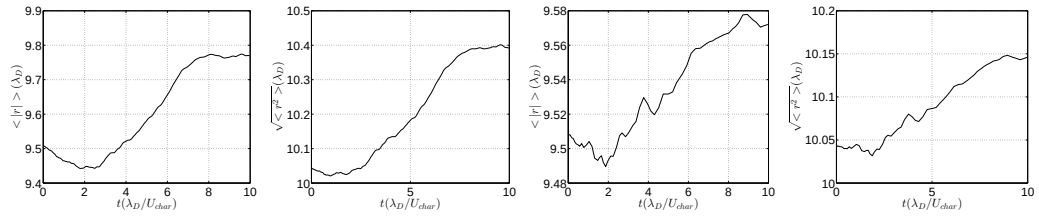
The cases with total vorticity density $1/N \sum_i \gamma_i \approx 0$ (2 in the numbered list) are presented in Figures 7.9-7.14. Generally these show fast ballistic expansion for all but the smallest time scales. This is consistent with the result shown in Figures 7.3 and 7.4. Again, we would expect the expansion to become diffusive at larger t.

For reference and comparison we can refer to the case *with* finite inertia, as in a classical gas with collisional interactions between atoms and molecules. In this case an initial condition with a compact cloud would expand as a sound wave. For this case we would find an expanding ring (expanding like $R \sim c_0 t$) in our plane, a finding completely different from e.g. Figures 7.1 and 7.3.



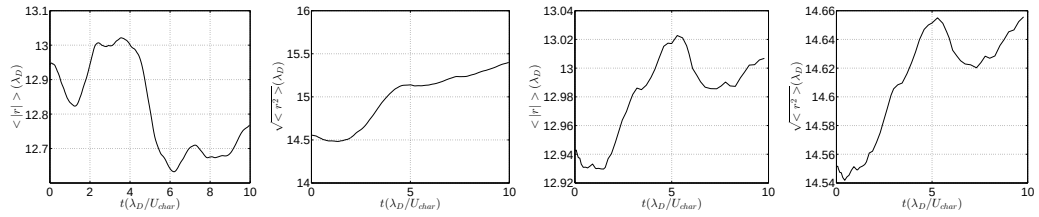
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.6: *Mean expansion (a,c) and root mean square expansion (b,d) of an elliptic cloud of line charges, with and without shielding, all with line charge density* $\gamma = 1$



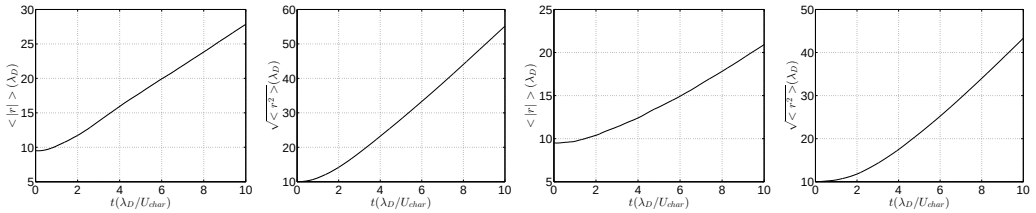
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.7: *Mean expansion (a,c) and root mean square expansion (b,d) of a circular cloud of line charges, with and without shielding, all with line charge density uniformly distributed such that* $\gamma \in (0, 1]$



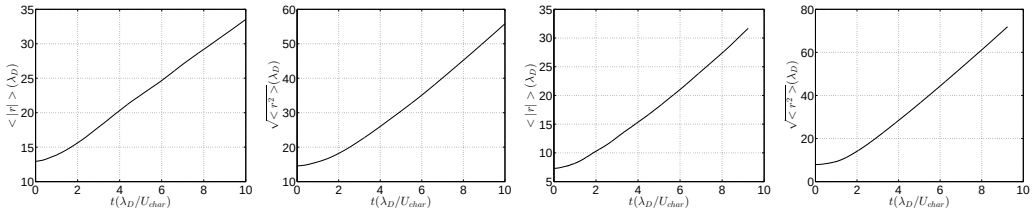
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.8: *Mean expansion (a,c) and root mean square expansion (b,d) of an elliptic cloud of line charges, with and without shielding, all with line charge density uniformly distributed such that* $\gamma \in (0, 1]$



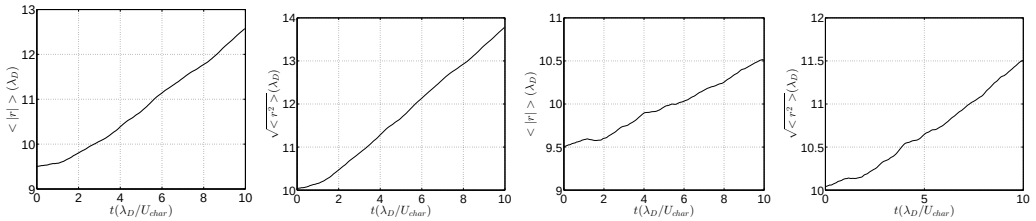
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.9: *Mean expansion (a,c) and root mean square expansion (b,d) of a circular cloud of line charges, with and without shielding, all with line charge density randomly distributed to* $\gamma = \pm 1$



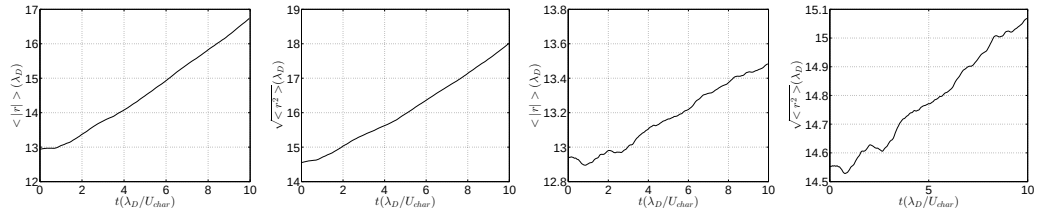
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.10: *Mean expansion (a,c) and root mean square expansion (b,d) of a elliptic cloud of line charges, with and without shielding, all with line charge density randomly distributed to* $\gamma = \pm 1$



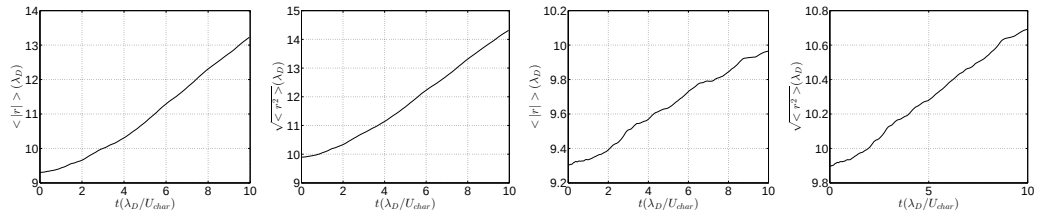
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.11: *Mean expansion (a,c) and root mean square expansion (b,d) of a circular cloud of line charges, with and without shielding, all with line charge density uniformly distributed so that that* $\gamma \sim U(-1, 1)$



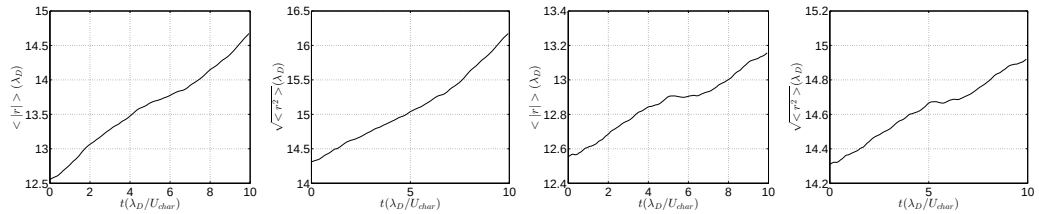
(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.12: *Mean expansion (a,c) and root mean square expansion (b,d) of a elliptic cloud of line charges, with and without shielding, all with line charge density uniformly distributed so that that* $\gamma \sim U(-1, 1)$



(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.13: *Mean expansion (a,c) and root mean square expansion (b,d) of a circular cloud of line charges, with and without shielding, all with line charge density* $\gamma \sim N(0, \frac{1}{2})$



(a) *Unshielded* (b) *Unshielded* (c) *Shielded* (d) *Shielded*

Figure 7.14: *Mean expansion (a,c) and root mean square expansion (b,d) of a elliptic cloud of line charges, with and without shielding, all with line charge density* $\gamma \sim N(0, \frac{1}{2})$

7.2 Collisions of four vortex clouds

We have simulated the collision between four circular "clouds" of vortices, with two different radii, arranged similarly to the collisions discussed in Section 3.2.3. These single vortex collisions are the extreme limit of cloud collisions when the ratio of the cloud radius and the separation distance between clouds $r/d \to 0$. It can be interesting to note that a single pair will propagate in the same way as two point vortices, and that in a simulation of a single pair, the individual clouds keep their separate identities for at least $500t_0$.

Samples of the space-time evolution of cloud pairs for selected parameters b and d (see Figure 7.15) are shown in Figures 7.16-7.19. Extracts of parameters characterizing the final states are summarized in Tables 7.2 and 7.3. These tables contain a wealth of information, which we believe can most readily be interpreted in terms of collisional cross sections. A part of the data are extracted in Figure 7.20

In the simulation we held $d = 20$ fixed while we have varied the impact parameter b from 0 to $2d$. This has been done for two different cloud radii, $r_1 = 1/2\sqrt{200} \approx 7.07107$ and $r_2 \approx \sqrt{200} = 14.14214$ while the total number of individual vortices was held constant at 401, i.e. 100 vortices in each cloud, except for cloud 1, which for programming reasons has 101 individual vortices.

This system is a generalisation of the collision of two vortex pairs, discussed in section 3.2.3, to distributions of vorticity. The point vortex collision is the limit where $r/d \to 0$. The greatest difference between the point vortex limit and the ensemble systems is the increasing number of degrees of freedom in the ensemble collisions. There is the centre of vorticity motion of each ensemble, the motion of individual line vortices around the ensemble centre of vorticity, as well as the possibility of deformation and splitting of the ensembles.

The results of the simulations with ensemble radius $r_1 = 1/2\sqrt{200} \approx 7.07107$ for varying impact parameter b are presented in Figures 7.16 and 7.17 and Table 7.2. Immediately we can see that the general behaviour can be classified into several types, depending on b.

1. "Elastic" collisions, no excitation of other degrees of freedom than the centre of vorticity motion. The vortex pairs interact and change partners in the same way as the individual vortices in section 3.2.3. This can also be classified as collisions with deflections of 90 deg or more. For $b = 0$ to $b = 0.7d$, Figures 7.16(a) and 7.16(b).

2. Almost elastic collisions, but with slight "inelaticity", i.e. a few line vortices are separated from the original ensemble, or are exchanged

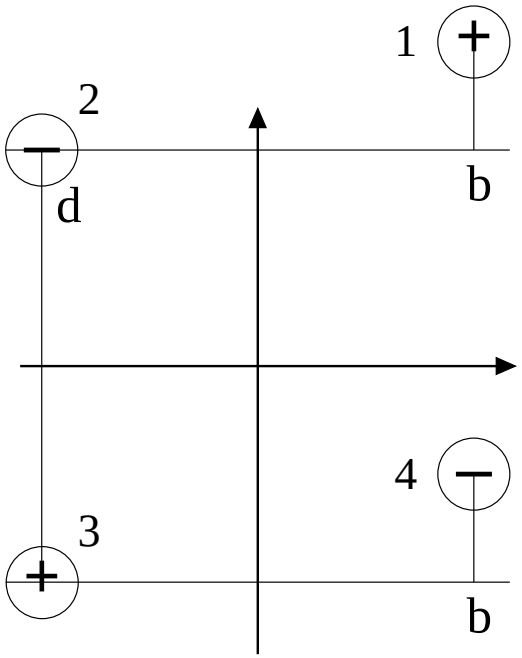


Figure 7.15: *The initial placement of the vortex clouds in the collision simulations. The +/− signs give the polarity of the cloud, d is the distance between the two vortices in the pair, and b is the impact parameter, which is a shift in the position of this pair. The clouds are numbered counter-clockwise after the quadrant of the plane in which they have their initial position for* $b = 0$.

between clouds. For $b = 0.75d$ to $b = 0.8d$, Figures 7.16(c) and 7.16(d).

3. Ensemble 2 and 4 connect slightly, but split again. The two new pairs exit in quadrants 1 and 3. Many vortices are exchanged between 2 and 4. For $b = 0.85d$. Not shown.

4. Inelastic collisions resulting in a bound state. Two ensembles with the same polarity merge and produce a compound vortex of approximately double strength. The two other two ensembles orbit the central combined vortex. The result is a stable tripolar vortex configuration resembling the tripolar vortices found experimentally and described by Vranjes et al. (2002), Okamoto et al. (2003) and Okamoto et al. (2005). For $b = 0.9d$ to $b = 1.25d$. The interesting observation here is that such "bound states" of three vortex clouds can result from an inelastic collision of four vortex clouds. Figures 7.16(e) and 7.16(f).

5. Strong interaction of the two ensembles with the same polarity resulting in the formation of a combined central vortex, but the tripolar configuration is not stable, and the central vortex splits into two approximately equal ensembles again. The result is the re-emergence of two pairs of ensembles. For $b = 1.3d$ to $b = 1.5d$, Figures 7.17(a) and 7.17(b).

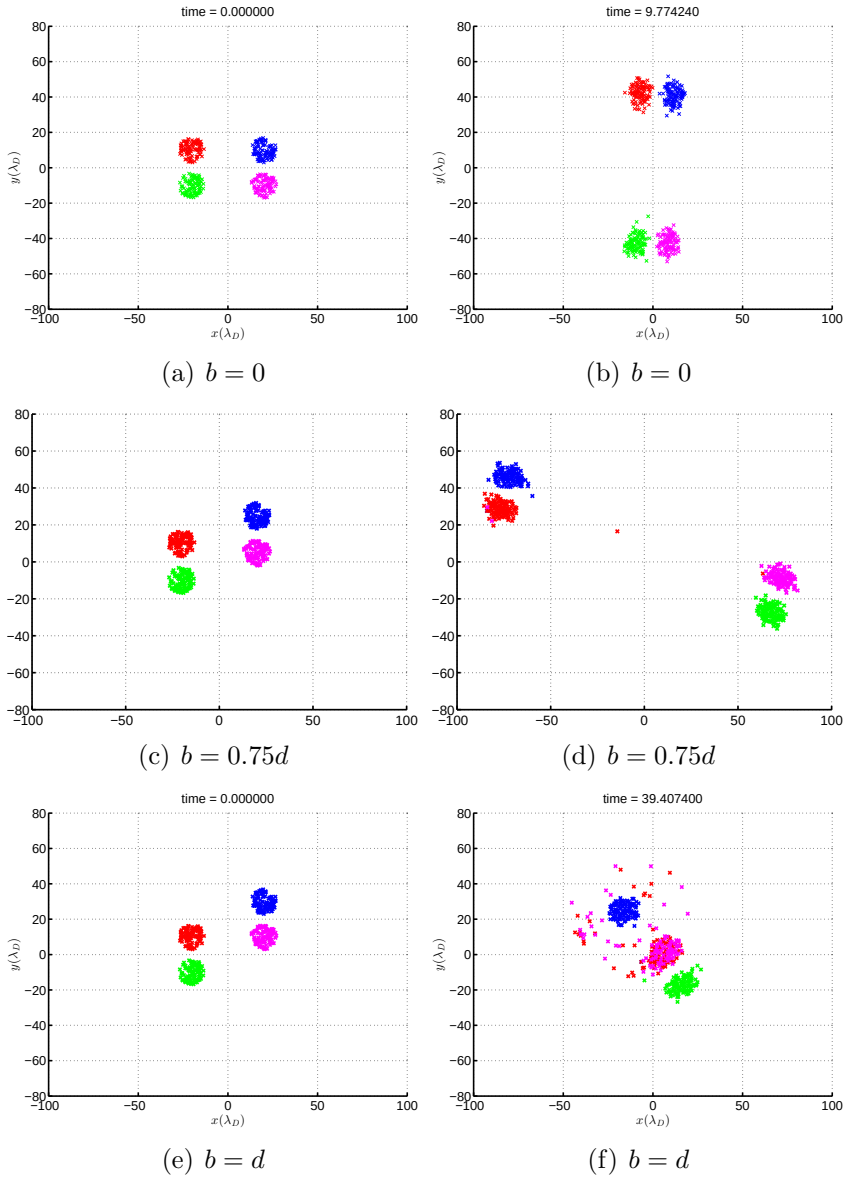
6. A region of weaker interactions, with a few line vortices exchanged, but otherwise little deflection of the centre of vorticity motion. For $b = 1.55d$ to $b = 1.95d$, Figures 7.17(c) and 7.17(d).

7. Slight deflection of the centre of vorticity motion, but otherwise little interaction. For $b = 2d$, Figures 7.17(e) and 7.17(f).

8. When $b \gg d$ interactions will become negligible. Not shown.

The doubling of the ensemble radius, to $r \approx \sqrt{200}$, while holding all other parameters constant results in significantly altered behaviour. The results are presented in Figures 7.18 7.19 and Table 7.3. First we see that doubling the radius means that there is an overlap between the component ensembles of one pair and the density of line vortices is now reduced by a factor 4.

As shown in the figures, there is now no stable bound state, in contrast with the result for clouds with half the radius. The individual line vortices are more lightly bound to the ensemble of which they are part, and thus the exchange and ejection of individual vortices happens for all values of b. The overall interaction between the centres of vorticity is less strong, and the behaviour is in all cases less like the point vortex collisions.

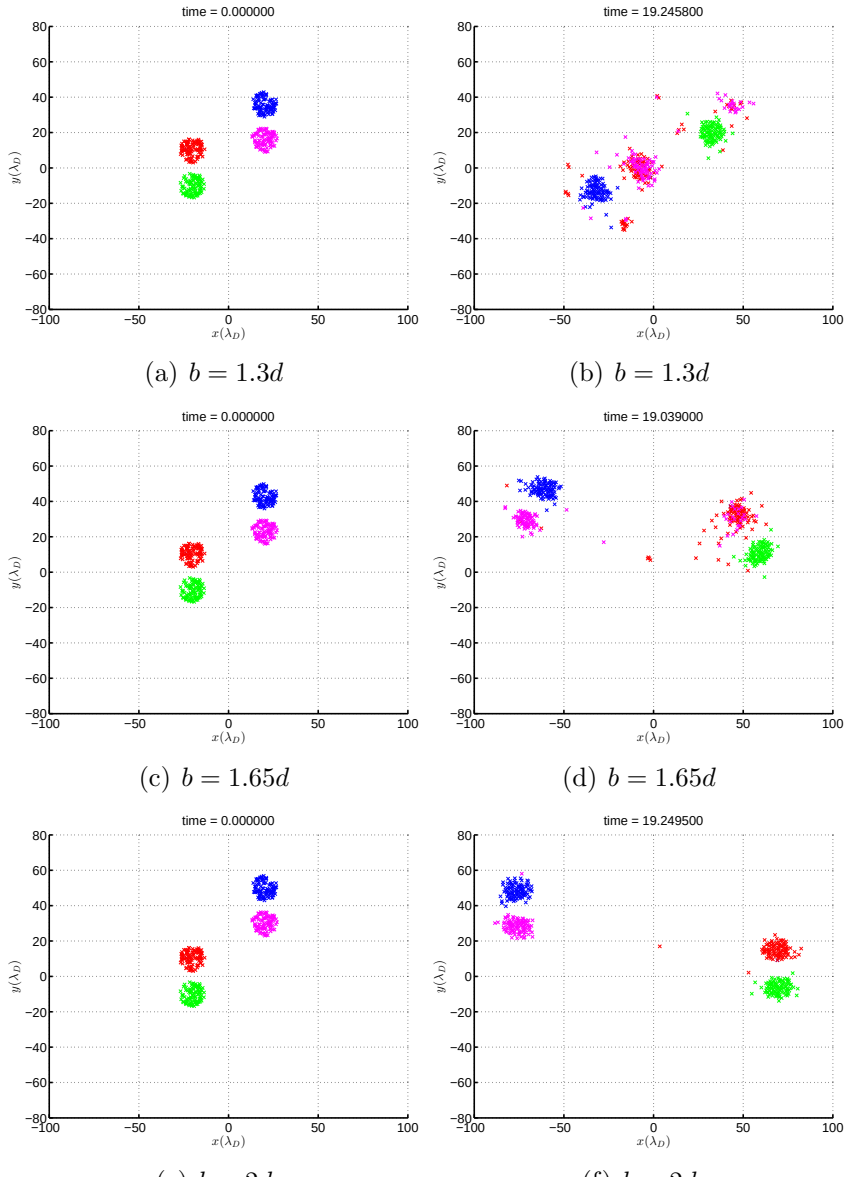


(a) $b = 0$ (b) $b = 0$

(c) $b = 0.75d$ (d) $b = 0.75d$

(e) $b = d$ (f) $b = d$

Figure 7.16: *Collision of four vortex clouds with radius* $r = 1/2\sqrt{200}$ *and 100 individual vortices in each ensemble. Each row of figures show the initial positions of the four vortex clouds (left) and the state of the system at the end of the simulations (right) for one impact parameter* b*. Here* $b = 0, 0.75d, d$*.*



(a) $b = 1.3d$ (b) $b = 1.3d$

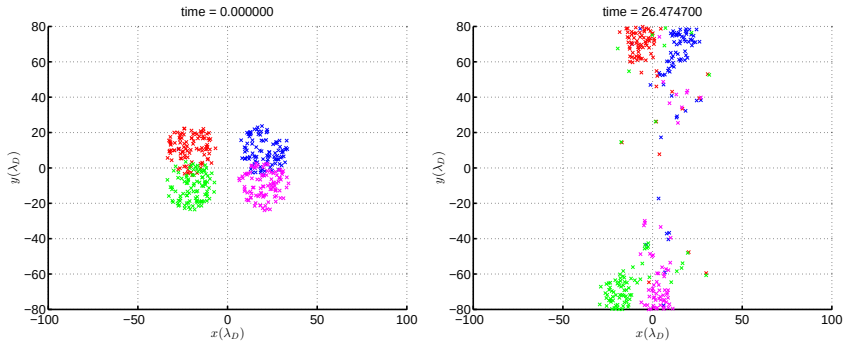
(c) $b = 1.65d$ (d) $b = 1.65d$

(e) $b = 2d$ (f) $b = 2d$

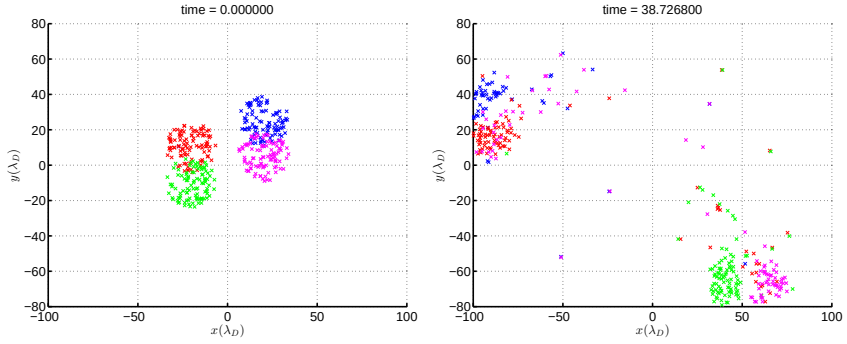
Figure 7.17: *Collision of four vortex clouds with radius* $r = 1/2\sqrt{200}$ *and 100 individual vortices in each ensemble. Each row of figures show the initial positions of the four vortex clouds (left) and the state of the system at the end of the simulations (right) for one impact parameter* b*. Here* $b = 1.3d, 1.65d, 2d$*.*

The centre of vorticity for the entire system remains fixed in all cases in agreement with the analytical result (3.21)-(3.22).

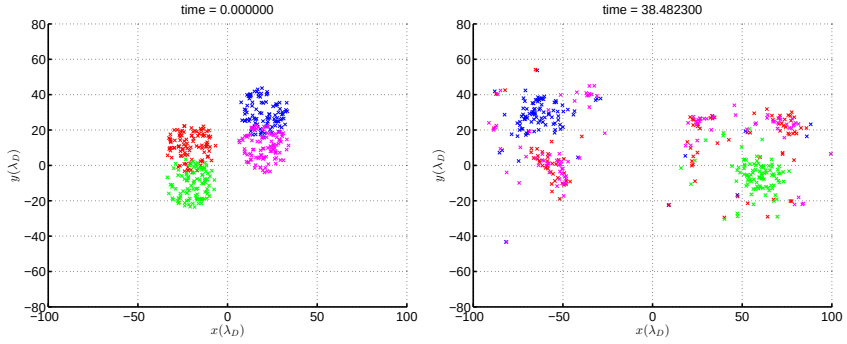
We can interpret Figure 7.20 as an illustration of collision cross sections of moving compact vortex cloud pairs, here considered as moving macroscopic particles. For a dense cloud (blue in the figure) we find all collisions with $b < 0.85d$ to be elastic, and no individual vortices are left in the central part (see Table 7.2 and 7.3). Similarily for $b > 1.35$. For $0.85 < b < 1.35$, however, we have a range of impact parameters leading to nearly completely inelastic collisions. This may be a somewhat counter-intuitive result.



(a) $b = 0$ (b) $b = 0$

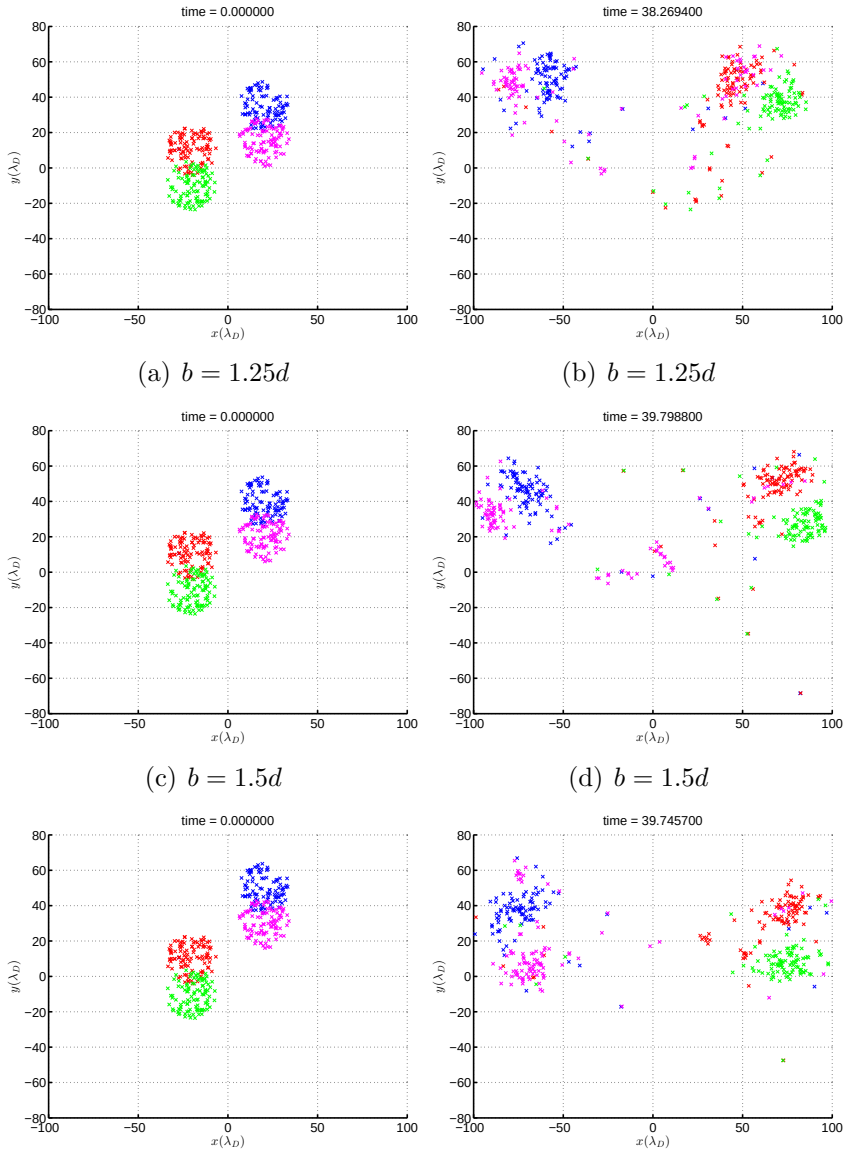


(c) $b = 0.75d$ (d) $b = 0.75d$



(e) $b = d$ (f) $b = d$

Figure 7.18: *Collision of four vortex clouds with radius* $r = \sqrt{200}$ *and 100 individual vortices in each ensemble. Each row of figures show the initial positions of the four vortex clouds (left) and the state of the system at the end of the simulations (right) for one impact parameter* b*. Here* $b = 0, 0.75d, d$*.*



(a) $b = 1.25d$ (b) $b = 1.25d$

(c) $b = 1.5d$ (d) $b = 1.5d$

(e) $b = 2d$ (f) $b = 2d$

Figure 7.19: *Collision of four vortex clouds with radius* $r = \sqrt{200}$ *and 100 individual vortices in each ensemble. Each row of figures show the initial positions of the four vortex clouds (left) and the state of the system at the end of the simulations (right) for one impact parameter* b*. Here* $b = 1.25d, 1.5d, 2d$*.*

Table 7.2: *The table lists the number of individual point vortices in each quadrant of the plane, and in a circular central region with radius $R = 10$, at the last time step in the simulation with impact parameter b. The circular ensembles have radius $r = 1/2\sqrt{200}$. In each cell of the table the nested tabular lists the total number of vortices in that region based on which of the clouds, numbered as shown in Figure 7.15, the vortex originated from.*

b	1st quadrant	2nd quadrant	3rd quadrant	4th quadrant	Centre
0	0 100 0 0	100 1 0 0	0 0 100 0	0 0 0 100	0 0 0 0
5	0 0 0 0	100 101 0 0	0 0 0 0	0 0 100 100	0 0 0 0
10	0 0 0 0	100 101 0 0	0 0 0 0	0 0 100 100	0 0 0 0
15	0 0 0 0	99 101 0 2	0 0 0 0	1 0 100 98	0 0 0 0
16	3 0 0 97	97 101 0 3	0 0 0 0	0 0 100 0	0 0 0 0
17	29 0 100 69	0 0 0 0	71 101 0 31	0 0 0 0	0 0 0 0
18	10 0 0 12	2 0 100 4	16 1 0 10	4 100 0 5	68 0 0 69
19	4 0 0 6	3 0 100 5	28 0 0 24	25 100 0 22	40 1 0 43
20	0 0 0 4	25 100 0 28	12 1 3 6	0 0 97 0	63 0 0 62
21	20 0 15 21	14 0 85 18	9 9 0 6	4 91 0 8	53 1 0 47
22	3 0 37 7	4 0 63 11	19 56 0 13	17 45 0 11	57 0 0 58
23	9 0 100 17	18 1 0 18	12 98 0 2	3 2 0 6	58 0 0 57
24	5 0 100 20	4 0 0 3	30 100 0 7	6 0 0 3	55 1 0 67
25	12 0 100 23	24 0 0 17	11 100 0 8	2 0 0 2	51 1 0 50
26	20 0 100 28	2 0 0 3	20 100 0 5	4 0 0 7	54 1 0 57
30	85 1 99 31	13 100 0 68	2 0 0 1	0 0 1 0	0 0 0 0
35	97 1 35 7	2 100 0 93	0 0 0 0	1 0 65 0	0 0 0 0
40	100 1 1 0	0 100 0 100	0 0 0 0	0 0 99 0	0 0 0 0

Table 7.3: *The table lists the number of individual point vortices in each quadrant of the plane, and in a circular central region with radius $R = 10$, at the last time step in the simulation with impact parameter b. The circular ensembles have radius $r = \sqrt{200}$. In each cell of the table the nested tabular lists the total number of vortices in that region based on which of the clouds, numbered as shown in Figure 7.15, the vortex originated from.*

b	1st quadrant	2nd quadrant	3rd quadrant	4th quadrant	Centre
0	11 86 6 8	84 3 4 1	2 5 83 85	3 7 7 6	0 0 0 0
5	1 1 1 2	88 91 6 11	2 1 13 2	9 8 79 82	0 0 1 3
10	3 3 0 2	84 95 4 22	0 0 0 0	13 3 96 75	0 0 0 1
15	2 2 2 4	76 96 1 39	0 2 0 2	22 1 97 55	0 0 0 0
16	1 3 2 5	58 96 0 36	19 2 1 6	22 0 97 53	0 0 0 0
17	2 1 4 3	43 96 0 34	29 3 1 16	26 1 95 47	0 0 0 0
18	2 3 2 7	33 88 1 40	27 8 2 14	38 2 95 39	0 0 0 0
19	18 5 3 10	8 90 0 34	45 5 1 29	29 1 95 27	0 0 1 0
20	50 5 22 38	16 92 0 36	23 2 0 22	11 2 78 4	0 0 0 0
21	63 5 90 40	22 93 0 47	2 1 0 4	12 2 10 4	1 0 0 5
22	66 2 89 20	12 97 0 56	9 2 0 12	5 0 11 0	8 0 0 12
23	84 7 94 43	5 92 1 41	6 1 0 15	5 1 5 1	0 0 0 0
24	78 5 92 26	2 91 1 43	13 4 0 30	7 1 7 1	0 0 0 0
25	86 7 90 33	6 94 3 61	1 0 1 4	7 0 6 2	0 0 0 0
26	85 6 93 26	5 95 1 70	2 0 1 3	8 0 5 0	0 0 0 1
30	94 6 93 21	1 91 2 67	1 2 1 9	4 1 3 0	0 1 1 3
35	94 5 90 7	5 93 5 67	1 2 2 25	0 1 3 1	0 0 0 0
40	91 7 90 8	5 91 3 76	1 2 1 15	3 1 6 1	0 0 0 0

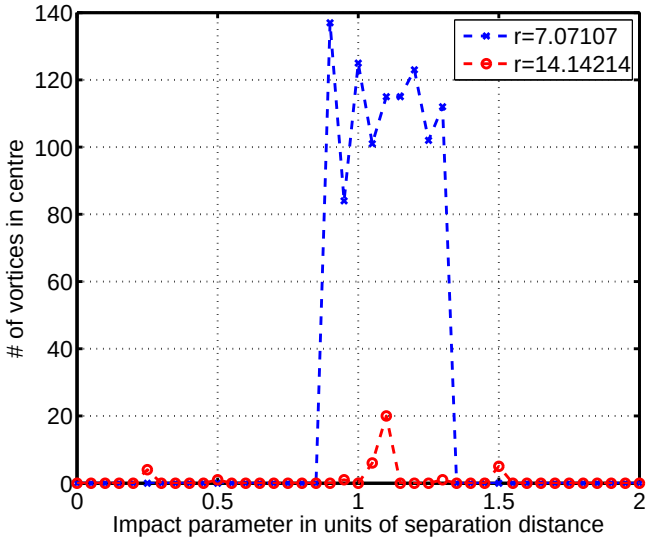


Figure 7.20: *The total number of individual point vortices in the central area as function of impact parameter, taken from the tables above. A high number of central vortices signifies a bound state, i.e. an inelastic collision of the four clouds. For the sequence of simulations with* $r = 14.14214$ *there was no bound state for any impact parameter.*

7.3 Homogeneous systems

For a description of the boundary conditions, see section 5.8

7.3.1 Constant vortex strengths

We have simulated a homogeneous system using periodic boundary conditions and image cells as an approximation. We did several simulations with the same initial conditions with different random numbers used to generate the initial uniform spatial distribution of vortices, where the position of vortex i was (x_i, y_i), $\quad x, y \in (-10, 10)$. The total vorticity in all these simulations was zero and $P(\gamma_i = 0.1) = P(\gamma_i = -0.1) = 1/2$ for vortex i. The number of image cell layers was set to 4, see section 5.8, and the boundaries of the unit cell was the same as the initial spatial distribution. Also, the number of vortices simulated was $N_s = 400$ with $N_t = N_e = N_s/10 = 40$ test vortices and Eulerian points.

We ran one simulation with $t_{max} = 100t_0$ and 13 with $t_{max} = 30t_0$. The simulation time varied due to the different random numbers drawn in each simulation. If the numbers drawn were such that the velocities experienced by the particles are high, the time step will be small, and the simulation will be slow. In this, one vortex pair, separated by a particularly short distance, can dominate since the shortest separation distance and highest velocity are used to calculate the time step.

In all experiments we saved the velocity components of the test particles and the components of the velocity field at the Eulerian points at every time step. We used these as the raw data for statistical analysis.

The raw and analysed data from the single simulation with $t_{max} = 100t_0$ is shown in Figures 7.21-7.30. Typical raw velocity data is shown in Figure 7.21. Figures 7.21(a) and 7.21(b) shows raw Lagrangian and Eulerian velocity data, respectively, of one vortex and fixed point for the entirety of the simulation.

The Lagrangian raw data shows a mean velocity close to zero, large variations over time-scales of a few t_0 with some sharp spikes, and many small scale fluctuations. That the mean velocity is close to zero is as expected from a fluctuating velocity field generated by an ensemble with net vorticity zero, and is consistent with the assumptions done in Chapter 4 on turbulent velocity fields. The large time scale variations in the velocity results from the overall interaction of the ensemble, sampled along the test vortex trajectory, while the small scale fluctuations are due to rapidly changing interactions between nearby vortices. The changes can be arbitrarily rapid since the vortices are without inertia. If the velocity field changes, a vortex will instantaneously adopt the new velocity.

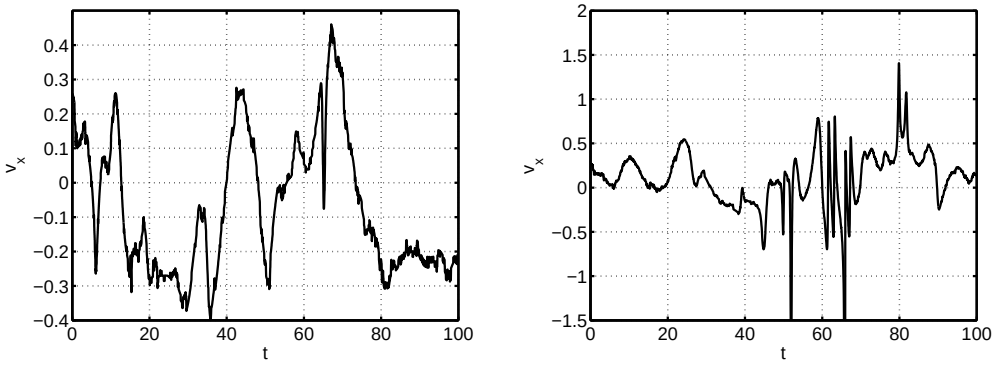
The Eulerian raw data show the same features: The near zero mean, long time scale variations and short fluctuations. In addition there are large spikes in the velocity data. Here the long scale variations and short scale fluctuations are due to the same mechanism. The velocity data here is just the value of the velocity field sampled at a fixed point, so there are again no inertial effects. The large spikes appear when one or more vortices come very close to the Eulerian fixed point. The probability of this happening with a test vortex is much lower since the vortex is not stationary. For the fixed point Eulerian sampling any vortex in the flow can come arbitrarily close to the observation point and consequently very high velocities can be detected. For the Lagrangian sampling, on the other hand, the conservation of the Hamiltonian imposes restrictions on the smallest separation of any two vortices, and therefore the detected velocity excess will be limited.

In Figure 7.22 we show the estimated probability density function of the Lagrangian (left) and Eulerian (right) velocities, averaged over 40 test vortices and fixed points, and again with x-component in blue and y-component in red. The dashed lines are normal distributions with mean 0 and the appropriate mean square velocity component as variance. This is plotted with logarithmic $y-$axis.

The PDFs generally show quite good agreement with the normal distribution in the areas around the mean, i.e. zero. The deviation from the normal distribution is much more marked farther from the mean value, and generally the estimated PDFs fall off less steeply than a normal distribution. This is much more marked in the Eulerian PDFs, which show quite heavy tails, and also significant probability for some very large values in some of the simulations. This is probably due to the effect noted above that a vortex can come much closer to one of the fixed points; this will lead to larger probability for large velocities.

Figure 7.23 shows estimates of the Lagrangian (left) and Eulerian (right) velocity autocorrelation functions for the v_x (blue) and v_y (red) components. These are the average correlation functions of 40 individual test vortices and fixed points. Eulerian and Lagrangian integral time scales can be calculated from this by the integral defined in equation (4.13) or, equivalently in the ideal case with infinitely long time series, from the power spectrum $S_L(f = 0)$ at frequency zero.

The Lagrangian autocorrelation shows, for the first 10-15 time units t_0, an approximately linear decrease on a plot with logarithmic y-axis; this corresponds to approximately exponential decrease. The Eulerian autocorrelation function on the other hand shows a linear trend on a log-log plot. This corresponds to polynomial decrease on linear axes. From this we can immediately see that the Lagrangian correlation times will be shorter than the Eulerian.



(a) *Lagrangian velocity raw data* (b) *Eulerian velocity raw data*

Figure 7.21: *Lagrangian and Eulerian raw data taken from the x-component velocity of one test vortex and one Eulerian point. The data are taken from the simulation with $\gamma = \pm 10.1$ from section 7.3.2.*

The autocorrelation functions for $t \gtrsim 20$ are mostly noise close to zero. Note that the velocity cross correlation functions (not shown) were close to zero for all times.

Figure 7.24 shows the Lagrangian (left) and Eulerian (right) power spectra, calculated by Fourier transform of the average autocorrelation functions, in a log-log plot. The dashed black line in the Lagrangian power spectrum is the line f^{-2}, where f is the frequency in units of $1/t_0$, and corresponds to an autocorrelation function that is an exact exponential function. Any deviation from this line is a deviation in the autocorrelation function from an exponential. For the very low frequencies, there is significant deviation from the f^{-2} line, apart from the area around $f = 1/t_0$ where the correspondence is good. For lower frequencies the power spectrum falls off less rapidly; for higher frequencies the data becomes noisy, but the trend is more rapid decrease in power as function of frequency.

The dashed black line in the Eulerian power spectrum is the line f^{-1}. Except for the very low frequencies the power spectrum shows very good correspondence with this trend. The dependence on frequency is, however, slightly steeper than f^{-1}, and this is good, since a power spectrum $\propto f^{-1}$ would lead to diverging total power, and mean frequency $\langle f \rangle \to \infty$ if we extrapolate to a non band-limited signal.

We will now present results that are obtained from comparing the 15 simulations with identical initial conditions. In the following we compare the Eulerian and Lagrangian mean square velocities (Figure 7.25) and integral time scales (Figure 7.26).

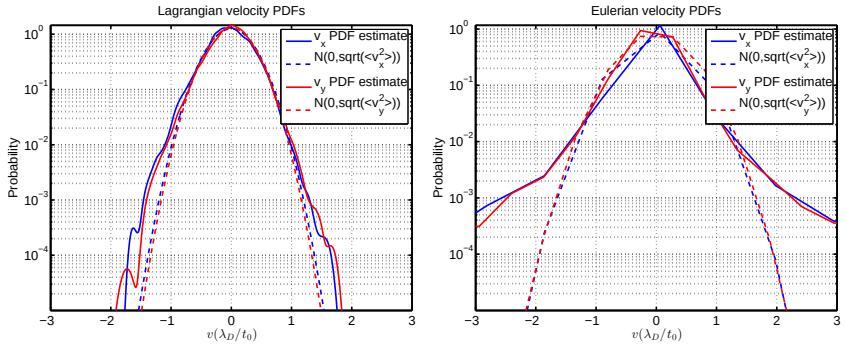


Figure 7.22: *Lagrangian(left) and Eulerian (right) velocity probability density functions.*

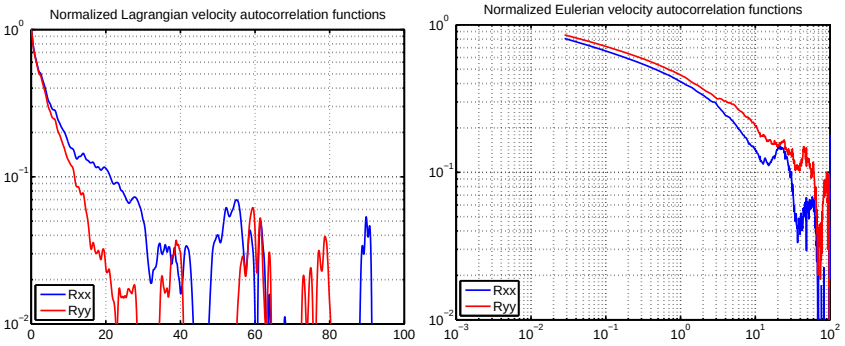


Figure 7.23: *Lagrangian (left) and Eulerian (right) velocity autocorrelation functions*

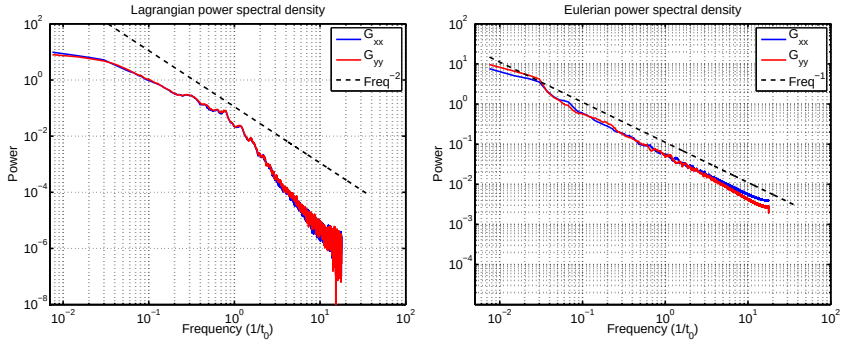


Figure 7.24: *Lagrangian (left) and Eulerian (right) power spectra.*

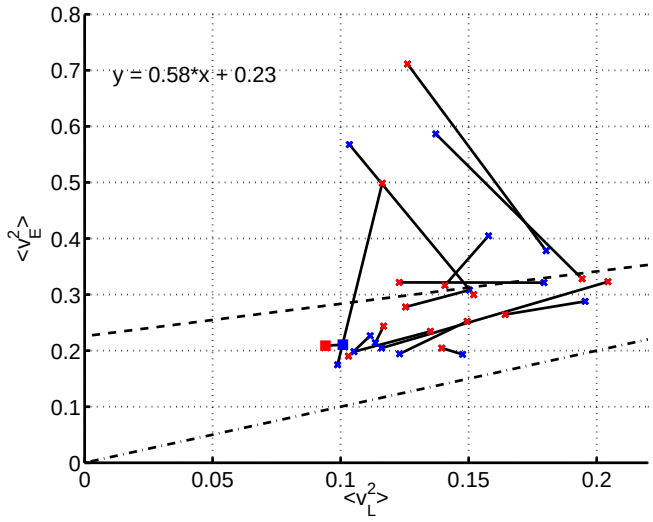


Figure 7.25: *Eulerian and Lagrangian mean square velocity components plotted as* $(\langle v_L^2 \rangle, \langle v_E^2 \rangle)$. *The blue markers are the x-components and the red are the y-components. Each pair connected with a black line is the result of a single simulation. The single pair marked with squares is the simulation with* $t_{max} = 100t_0$. *The dashed line is the best linear fit to the data with the equation shown in the figure. The dot-dashed line is the line* $\langle v_L^2 \rangle = \langle v_E^2 \rangle$.

We have previously shown that the Lagrangian and Eulerian mean square velocities will be the same for a homogeneous, isotropic and incompressible system (Section 4.2.2). A comparison between the Lagrangian and Eulerian mean square velocities as calculated from our simulations is shown in Figure 7.25. The blue and red marks connected with a black line are the points $(\langle v_{xL}^2 \rangle, \langle v_{xE}^2 \rangle)$ and $(\langle v_{yL}^2 \rangle, \langle v_{yE}^2 \rangle)$, respectively, for a single simulation. The dashed line is the best linear fit to all the data points, with equation $\langle v_E^2 \rangle = 0.6 \langle v_L^2 \rangle + 0.22$. This result diverges significantly from what we would expect from our discussion of mean square velocities in Chapter 4. There is a possibility that this might be a result of inhomogeneities and anisotropies in the system, and running the simulation with higher t_{max} might give the system time to become homogeneous and isotropic. Note that the simulation with $t_{max} = 100t_0$ shows one of the smallest anisotropies. The discrepancy might also result from the small sample size of 15 simulations.

In section 4.2.1 we discussed the problem of relating Eulerian and Lagrangian integral time scales (τ_E and τ_L). In many applications it is of great interest to be able to find a transformation relating the two time scales. Because of the nature of our numerical simulations we can obtain both time

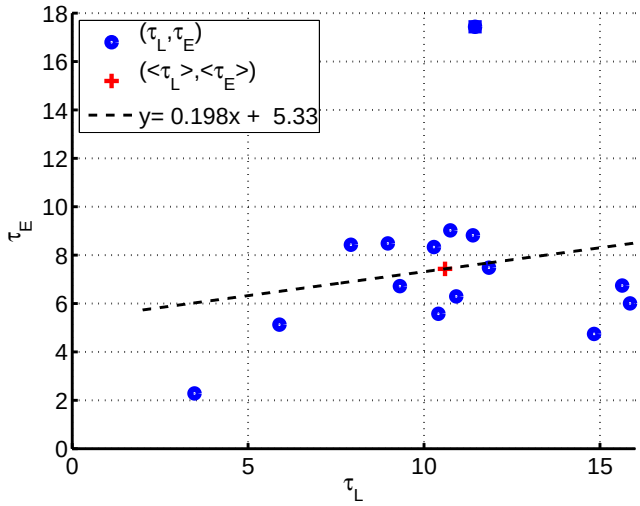


Figure 7.26: *Eulerian and Lagrangian integral time scales. Calculated as the average of the* $S_{xx}(f = 0)$ *and* $S_{yy}(f = 0)$ *for the two cases. Each of the points are* (τ_L, τ_E) *for one simulation. The square is the result from the one simulation with* $t_{max} = 100t_0$*. The dashed line is the best linear fit to the data with the equation shown in the figure.*

scales from the respective power spectra at frequency $f = 0$. The comparison is shown i Figure 7.26 where each blue circle is (τ_L, τ_E) for one simulation. The blue square is the result from the simulation with $t_{max} = 100t_0$. The red cross is the point $(\langle\tau_L\rangle, \langle\tau_E\rangle)$ and the dashed line is the best linear fit to the data with equation $\tau_E = 0.2\tau_L + 5.3$. There is significant spread of the data, and the sample size is again moderate consisting of 15 simulations.

7.3.2 Varying vortex strength

We ran five simulations with the same random numbers generated for every simulation and varying vortex strengths $|\gamma_j| = 0.1$, 0.05, 0.025, 0.0125, 0.00625. For simulation j all vortex strengths were randomly distributed to $\pm|\gamma_j|$ and the simulations were run to $t_{max} = 100t_0$. The one data set, from which the raw data was discussed above, is also included here as the $|\gamma_j| = 0.1$. All other parameters were identical to the simulations described in Section 7.3.1.

The first figure we present (Figure 7.27) shows the dependence of the root mean square velocity, which is a typical velocity of the system, on the magnitude of the vortex strengths. This shows a slightly lower than linear dependence in a log-log plot. The dashed line is $\propto |\gamma_j|$. Higher vortex

strengths will lead to stronger interactions in the system, and thus lead to higher root mean square velocities. Since the velocity at any radial position from the centre of an individual vortex is linearly proportional to γ, we find the linear dependence of $\sqrt{\langle v^2 \rangle}$ to be reasonable.

In Figure 7.28 we show the calculated Lagrangian integral time scale as function of vortex strength. τ_L was calculated by numerical integration of the autocorrelation functions (a) and through Fourier transform (b) as the component of the power spectrum at $f = 0$. In both panels the blue line corresponds to the time scale calculate from the x-components of the velocities and the red line to the y-components. The black lines are the average of the x and y-components and is what will be referred to as τ_L. The dependence is largely the same, but the time scales calculated from the power spectra are somewhat longer than the ones from numerical integration. Because of the finite length of the time series, the numerical integration of the autocorrelation function was stopped at the first time lag where R_{xx} or R_{yy} was equal to zero. This will tend to make the numerically integrated time scales artificially short.

The time scale τ_L generally decreases for increasing vortex strengths. This is as expected from the fact that the root mean square velocity increases with increasing vortex strength. Higher velocities will lead to a single vortex randomly interacting with a higher number of other vortices in a given time, which in turn will generally lead to shorter correlation times. The trend does not seem to be valid for the two highest vortex strengths, if we look at the y-component in (b). This is probably due to the value at $|\gamma| = 0.05$ being an outlier due to random fluctuations in the system.

The difference between the x- and y-component is a measure of the anisotropy in our distributions. We would expect that in an isotropic system, these would be equal and that if our simulation time $t_{max} \to \infty$ the x- and y-components will converge. There is no a priori reason to believe that the x-component should be larger than the y-component. This will vary with the random initial conditions, and every initialization of the system will be anisotropic to different degrees. Since τ_L depends on the root mean square velocity, we can conclude from Figure 7.25 that it will be equally likely that the x or y time scale will be the longest.

For every value of the vortex strength we have estimated the diffusion coefficient $D = 2 \langle v_L^2 \rangle \tau_L$ using the total mean square velocities and average time-scale τ_L. The result is shown in Figure 7.29 where (a) shows dependence on vortex strength and (b) shows dependence on root mean square velocity $\sqrt{\langle v_L^2 \rangle}$.

The dependence of the effective diffusion coefficient on the root mean square velocity is close to linear, while the dependence on vortex strength is

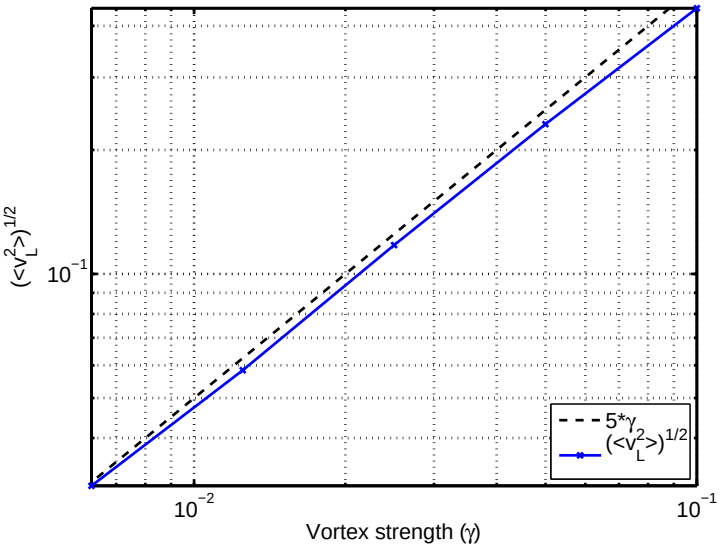
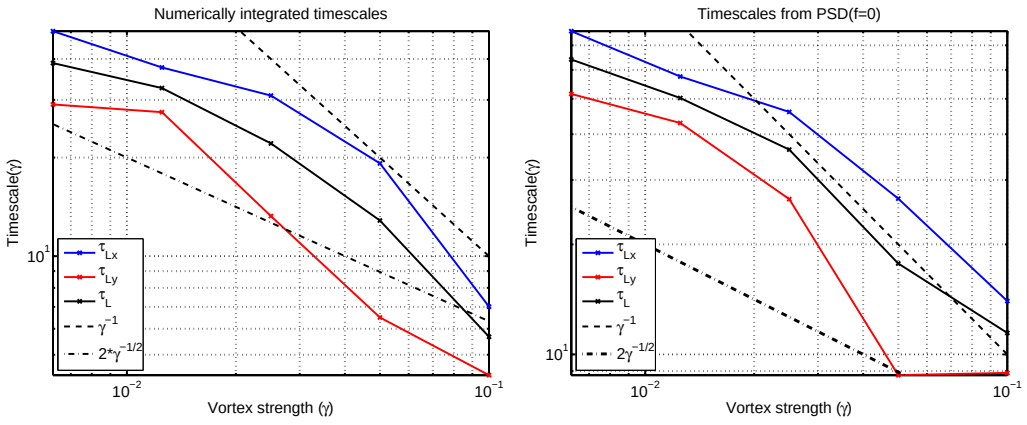
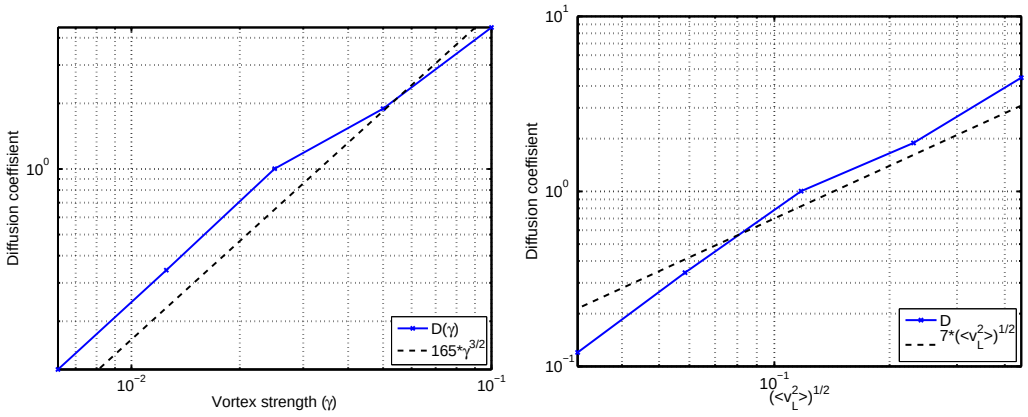


Figure 7.27: *The root mean square velocity as a function of normalized vortex strength*



(a) (b)

Figure 7.28: *Lagrangian integral timescales τ_L calculated by numerically integrating the autocorrelation functions (7.28(a)) and from $S(f = 0)$ (7.28(b))*



(a) (b)

Figure 7.29: *Estimated effective diffusion coefficients as function of normalized vortex strength (7.29(a)) and of root means square velocity (7.29(b))*

close to $\gamma^{3/2}$.

In classical diffusion theory the diffusion coefficient is proportional to the ratio of a mean free path to the power of two and a collision frequency, i.e. $D \propto l^2/\nu$, where $\nu \propto l/v$ where v is a typical velocity, in our case $\sqrt{\langle v^2 \rangle}$. If l is constant for different values of v the diffusion coefficient will be proportional to both v and l. We have already seen that $\sqrt{\langle v^2 \rangle} \propto \gamma$, and constant l would then lead to $D \propto \gamma$. From this we can conclude that l is not constant when γ is varied.

Equation 4.12 provides a way to calculate the root mean square displacement as a function of time $\sqrt{\langle r^2(t) \rangle}$. We have implemented this through numerical integration of the correlation functions, and the result is shown in Figure 7.30 for the different vortex strengths. The dashed lines are proportional to t and $t^{1/2}$. As we expect, for short times we see that $\sqrt{\langle r^2(t) \rangle} \propto t$, which is the ballistic limit, discussed in Chapter 4. Conversely, for large times, we see that $\sqrt{\langle r^2(t) \rangle}$ converges to $t^{1/2}$, which is the diffusion limit, where the behaviour of the system is entirely determined by the diffusion coefficient. In agreement with previous results, the time at which the diffusion limit can be said to be valid increases with decreasing vortex strengths.

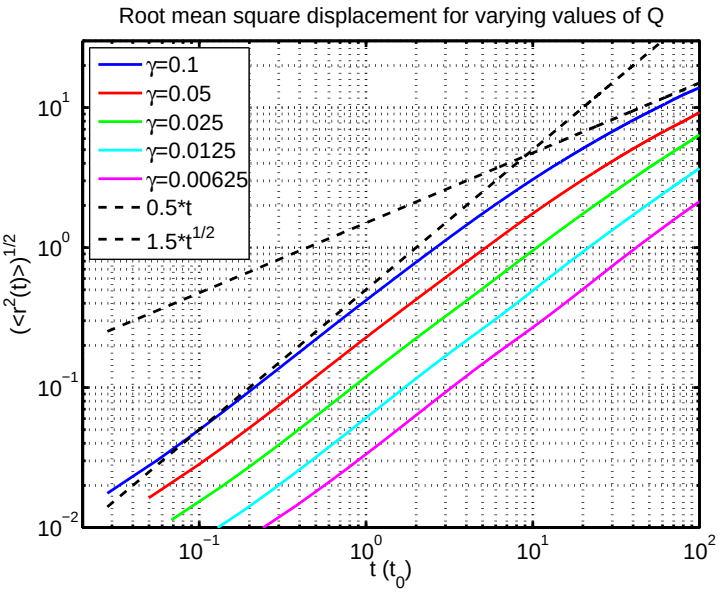


Figure 7.30: $\sqrt{\langle r^2(t)\rangle}$ *for different vortex strengths. The dashed lines with slopes 1 and* $1/2$ *are inserted for reference, giving the ballistic and diffusion limits respectively.*

Chapter 8

Discussions and Conclusions

Most of the results obtained have been discussed in the respective chapters and sections. The present chapter will only give a summary.

We have demonstrated that a many vortex system can evolve to have turbulence-like characteristics, such as turbulence-like correlation functions (Eulerian and Lagrangian) showing finite memory, continuous power spectra, etc. Also the basic features of turbulent transport are recovered. All this is found for computationally moderate requirements, i.e. our models typically contain a few hundred to a few thousand discrete vortices. The model also allows studies of large-scale coherent structures obtained by superimposing many discrete vortices. The present model is found to have randomly varying velocity fields as in turbulent neutral flows. One important difference is that classical turbulence is generally seen as a driven-dissipative system. For the present vortex system the Hamiltonian is given by the initial conditions and energy is neither dissipated nor injected.

We have analysed and demonstrated a number of basic properties of vortex systems and their numerical implementation.

1. In the implementation of the numerical model there were some areas of special concern:

    - Variable time step: To ensure that a vortex system is simulated correctly, careful attention must be paid to the time step used when solving the differential equations. We came to the conclusion that a variable time step, calculated in every iteration was the most efficient way to ensure adequate time resolution.
    - Periodic boundary conditions and image cells: The correct implementation of periodic boundary conditions and the handling of interactions with vortices in image cells presented a major challenge

- Subroutines for Bessel functions: The efficiency of any Bessel function subroutine is of great importance for large simulations. Highly optimized routines for calculating the modified Bessel functions should be a focus if large simulations of shielded vortices are of interest.

2. The basic properties of low dimensional vortex systems have been recovered through numerical simulations. We have demonstrated:
   - The propagation of vortex pairs for different combinations of vortex strengths.
   - Vortex pair collisions for different impact parameters.
   - The expansion/collapse of a vortex triplet given specific initial conditions.
3. We have studied the time evolution of large-scale coherent structures by superimposing many discrete vortices.
   - A vortex cloud of one polarity will have a long life time as a coherent structure. This is also valid for a pair of vortex clouds of opposite polarities.
   - A vortex cloud of approximately zero vorticity density will have short life time as a coherent structure.
   - Collisions of two vortex cloud pairs display the same basic dynamics as vortex pair collisions for small impact parameters. Established also parameter range for inelastic collisions.
4. The basic features of turbulent transport are recovered in terms of an approximately homogeneous and isotropic system of point vortices.
   - Realized the evolution of turbulent spectra for the velocity field.
   - Established phenomenological relations between Eulerian and Lagrangian integral time scales.
   - Identified ballistic and diffusive limits for turbulent diffusion of single particles.

8.1 Future perspectives

The time one has for the completion of a masters degree is necessarily limited. Due to this, many goals and interesting projects could not be completed. If

we get the opportunity to continue this study at a later time, the following changes and refinements would be prioritized:

- The implementation of Ewald summation methods for calculating the interactions in the periodic system. This will allow for a greater number of image cells without increasing the simulation time.
- The reformulation of the model by the use of "vortex-in-cell" methods (Leonard, 1980), similar to particle-in-cell methods. This will allow for a larger number of simulated vortices.
- The consideration of other vortex interaction functions F than the logarithmic an Bessel function interactions covered in this thesis. The vortices considered in this study are all singular for $r = 0$, other interaction functions can have the advantage of removing this singularity.
- The implementation of vortices of finite size.
- Incorporation of polarization drifts, i.e. the first higher order correction in the parameter $\langle\omega\rangle / \Omega_c$. This will be a very challenging problem analytically, and it has never been approached for vortex systems. (For model equations based on fluid theory this problem is much easier.)

Bibliography

M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover Publications, New York, 1965.

H. Aref. Motion of three vortices. *Physics of Fluids*, 22(3):393–400, 1979.

H. Aref. Integrable, chaotic and turbulent vortex motion in two-dimensional flows. *Annual Review of Fluid Mechanics*, 15:345–389, 1983.

G. K. Batchelor. Diffusion in a field of homogeneous turbulence. *Aust. J. Sci. Res.*, 2:437–450, 1949.

S. Corrsin. Estimates of the relations between eulerian and lagrangian scales in large reynolds number turbulence. *Journal of the Atmospheric Sciences*, 20(2):115–119, 1963.

William Crookes. The bakerian lecture: On the illumination of lines of molecular pressure, and the trajectory of molecules. *Philosophical Transactions of the Royal Society of London*, 170:135–164.

U. Frisch. *Turbulence: The Legacy of A. N. Kolmogorov*. Cambridge University Press, 1996.

W. Horton. Drift waves and transport. *Rev. Mod. Phys.*, 71:735–778, Apr 1999.

Y. Kimura. Similarity Solution of Two-Dimensional Point Vortices. *Journal of the Physical Society of Japan*, 56(6):2024–2030, 1987.

M. Kono, B. Krane, H. L. Pécseli, and J. Trulsen. Vortex dynamics in magnetized plasmas. *Phys. Scripta*, 58:238–245, 1998.

I Langmuir. Oscillations in ionized gases. *Proceedings of the National Academy of Sciences of The United States of America*, 14:627–637, 1928.

A Leonard. Vortex Methods for Flow Simulation. *Journal of Computational Physics*, 37(3):289–335, 1980.

P. C. Liewer. Measurements of Microturbulence in Tokamaks and Comparisons with Theories of Turbulence and Anomalous Transport. *Nuclear Fusion*, 25(5):543–621, 1985.

J. L. Lumley. Approach to Eulerian-Lagrangian Problem. *Journal of Mathematical Physics*, 3(2):309–&, 1962.

J. L. Lumley and H. A. Panofsky. *The Sructure of Atmospheric turbulence*. Interscience Publishers, 1964.

J. P. Lynov, A. H. Nielsen, H. L. Pécseli, and J. J. Rasmussen. Studies of the Eulerian-Lagrangian transformation in 2-dimansional random flows. *Journal of Fluid Mechanics*, 224:485–505, 1991.

M. Marvan. *Negative Absolute Temperatures*. Number 4 in Physics Paperbacks. Iliffe, London, 1966.

G. K. Morikawa and E. V. Swenson. Interacting Motion of Rectilinear Geostrophic Vortices. *Physics of Fluids*, 14(6):1058–1073, 1971.

A. Okamoto, K. Hara, K. Nagaoka, S. Yoshimura, J. Vranjes, M Kono, and M. Y. Tanaka. Experimental observation of a tripolar vortex in a plasma. *Physics of Plasmas*, 10(6):2211–2216, 2003.

A. Okamoto, K. Nagaoka, S. Yoshimura, J. Vranjes, S. Kado, M Kono, and M. Y. Tanaka. Tripolar vortex in a plasma. *IEEE Transactions on Plasma Sciences*, 33(2, Part 1):452–453, 2005.

R. K. Pathria. *Statistical Mechanics*. Butterworth-Heinemann, Oxford, 2 edition, 1998.

H. Poincaré. *Théorie des Tourbillons*. Editions Jacques Gabay, Paris, 1893.

H. L. Pécseli. *Fluctuations in Physical Systems*. Cambridge University Press, 2000.

P. H. Roberts. On the application of a statistical approximation to the theory of turbulent diffusion. *J. Math. and Mech.*, 6:781–799, 1957.

Yu. B. Rumer and M. Sh. Ryvkin. *Thermodynamics, Statistical Physics and Kinetics*. MIR Publishers, 1980.

H. Tennekes and J. L. Lumley. *A First Course in Turbulence*. The MIT press, Cambridge, Massachusetts, 1972.

J. Vranjes, A. Okamoto, S. Yoshimura, S. Poedts, M. Kono, and M. Y. Tanaka. Analytical description of a neutral-induced tripole vortex in a plasma. *Physical Review Letters*, 89(26), DEC 23 2002.

Appendix A

C++ source code

Listing A.1: main.cpp,

```
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Main file. Contains all the routines for calculating the velocities at the particle postitons, as well as initialization and data analysis routines. int main() does the iterating.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

#include "StringsandParticles.cpp"

#define XBINS 50
#define YBINS 50
#define HIST 2
#define PERIODIC
//#define DETERMINISTIC
#define TEST //if this is defined, it compiles the code which initializes test strings which are affected by others but does not affect other.

//#define BOUNDARY
//#define RESEED
//#define RANDOMSEED
#define BALANCED

//#define FOURCLOUD

int Np, Ns, Npmax, Nsmax, Nt, Ntmax;
Particles *p;
Strings *s;
Strings *test;

double *Xeuler;
double *Yeuler;

double pi = acos(-1);
//NB deklarere i main.cpp (globalt)
//double xu,xl,yu,yl;
double xl = -10.0;
double xu = 10.0;
double yl = -10.0;
double yu = 10.0;
double Rmaks = 1.5;
double epsilon0 = 8.85e-12;
//double K = 1/(2*pi*epsilon0)
double K = 1.0;
//Periodic boundary limits
double x_min = -10.0, x_max = 10.0, y_min = -10.0, y_max = 10.0;
double L = x_max-x_min;
int Mmax, Nmax;

double placeholder;
double leastdist;
int holder;

double dt;
double l_max;
double interval = 1e-3;
double nextstep = interval;
bool printtest = false;

double impact;

//INITIAL DISTRIBUTION PARAMETERS
int gauss;
double distx=10.0;
double disty=10.0;
#ifdef PERIODIC
bool circle = false;
#else
bool circle = true;
#endif



//Derivatives for particles
void dydt_part(double *y, double *dydt, double *a){
        dydt[0] = y[2];
        dydt[1] = y[3];
        dydt[2] = a[0];
        dydt[3] = a[1];
}

```

```
77  //Derivatives for strings
78  void dydt_string(double *y, double *dydt, double *v){
79          dydt[0] = v[0];
80          dydt[1] = v[1];
81  }
82
83  //Initializes the strings and particles with a filename and all values set to 0.
84  //Uses the constructor in class stringsandparticles
85  void initialize(double T, double dt){
86
87          Ns = Np = Nt = 0;
88          if(Npmax>0) p= new Particles[Npmax];
89          if(Nsmax>0) s= new Strings[Nsmax];
90  #ifdef TEST
91          if(Ntmax>0) test = new Strings[Ntmax];
92  #endif
93          double Q=0., B=0.;
94          double q=0.0, m=0.,u=0.,v=0.,x=0.,y=0.; //Charge, mass
95
96          string header="";
97  #ifdef PSPRINT
98          header="header.m";
99          ofstream mfil;
100         mfil.open(header.c_str(),ios::trunc);
101         mfil<<"Np="<<Npmax<<";"<<endl
102             <<"Ns="<<Nsmax<<";"<<endl;
103
104         mfil.close();
105 #endif
106
107         for(int i=0;i<Nsmax;++i){
108                 s[i].initialize(header, i, x,y,Q,B,dt,dydt_string);
109         }
110         for(int i=0;i<Npmax;++i){
111                 p[i].initialize(header,i,x,y,u,v,q,m,dt,dydt_part);
112         }
113 #ifdef TEST
114         for(int i=0;i<Ntmax;++i){
115                 test[i].initialize(header, i, x,y,Q,B,dt,dydt_string);
116         }
117 #endif
118
119 }
120
121 //Generates the strings and particles by making a arbitrary number of the active and
        fixing their values
122 void generate(int Npgen, int Nsgen, int Ntgen, bool initial){
123         //Initialize random function, dump first value
124         #ifdef RANDOMSEED
125         srand48(int(time(NULL))); drand48();
126         #else
127         srand48(1); drand48();
128         #endif
129         double Q=1., B=1.;
130         double q=0.0, m=1.; //Charge, mass
131         double x,y,u,v;
132         double Q_eff=0.0, Q_eff2 = 0.0;
133
134
135         if(Nsgen+Ns<=Nsmax){
136
137 #ifdef FOURCLOUD
138                 for(int i=0; i<Nsgen/4.0; ++i){
139                         if(initial){
140                                 double rad, angle, semi1, semi2;
141                                 semi1=1.0;semi2=1.0;
142                                 rad = 0.1*sqrt(distx*distx+disty*disty)*sqrt(drand48());
143                                 angle = 2*pi*drand48();
144                                 x = 20+semi1*rad*cos(angle);
145                                 y = impact+10+semi2*rad*sin(angle);
146
147                         }else{
148                                 x=0.0;
149                                 y=0.0;
150                         }
151                         Q=drand48();
152                         while(Q==0.0)Q=drand48();
153                         if(Q<0.0)Q = 1;
154                         else if(Q>0.0) Q = 1;
155                         else cout<<"DAFUQ"<<"_Q="<<Q<<endl;
156
157                         s[Ns++].setall(x,y,Q,B);
158                 }
```

```
159                     for(int i=Nsgen/4.0; i<Nsgen/2.0; ++i){
160                             if(initial){
161                                     double rad, angle, semi1, semi2;
162                                     semi1=1.0;semi2=1.0;
163                                     rad = 0.1*sqrt(distx*distx+disty*disty)*sqrt(drand48());
164                                     angle = 2*pi*drand48();
165                                     x = -20+semi1*rad*cos(angle);
166                                     y = 10+semi2*rad*sin(angle);
167
168                             }else{
169                                     x=0.0;
170                                     y=0.0;
171                             }
172                             Q=drand48();
173                             while(Q==0.0)Q=drand48();
174                             if(Q<0.0)Q = -1;
175                             else if(Q>0.0) Q = -1;
176                             else cout<<"DAFUQ"<<" Q="<<Q<<endl;
177
178                             s[Ns++].setall(x,y,Q,B);
179                     }
180                     for(int i=Nsgen/2.0; i<3*Nsgen/4; ++i){
181                             if(initial){
182                                     double rad, angle, semi1, semi2;
183                                     semi1=1.0;semi2=1.0;
184                                     rad = 0.1*sqrt(distx*distx+disty*disty)*sqrt(drand48());
185                                     angle = 2*pi*drand48();
186                                     x = -20+semi1*rad*cos(angle);
187                                     y = -10+semi2*rad*sin(angle);
188
189                             }else{
190                                     x=0.0;
191                                     y=0.0;
192                             }
193                             Q=drand48();
194                             while(Q==0.0)Q=drand48();
195                             if(Q<0.0)Q = 1;
196                             else if(Q>0.0) Q = 1;
197                             else cout<<"DAFUQ"<<" Q="<<Q<<endl;
198
199                             s[Ns++].setall(x,y,Q,B);
200                     }
201                     for(int i=3*Nsgen/4.0; i<Nsgen; ++i){
202                             if(initial){
203                                     double rad, angle, semi1, semi2;
204                                     semi1=1.0;semi2=1.0;
205                                     rad = 0.1*sqrt(distx*distx+disty*disty)*sqrt(drand48());
206                                     angle = 2*pi*drand48();
207                                     x = 20+semi1*rad*cos(angle);
208                                     y = impact-10+semi2*rad*sin(angle);
209
210                             }else{
211                                     x=0.0;
212                                     y=0.0;
213                             }
214                             Q=drand48();
215                             while(Q==0.0)Q=drand48();
216                             if(Q<0.0)Q = -1;
217                             else if(Q>0.0) Q = -1;
218                             else cout<<"DAFUQ"<<" Q="<<Q<<endl;
219
220                             s[Ns++].setall(x,y,Q,B);
221                     }
222 #else//
      ********************************************************************************************

223                     for(int i=0; i<Nsgen; ++i){
224                             if(initial){
225                                     if(circle){
226                                             double rad, angle, semi1, semi2;
227                                             semi1=2.0;semi2=0.5;
228                                             rad = sqrt(distx*distx+disty*disty)*sqrt(drand48
                                                    ());
229                                             angle = 2*pi*drand48();
230                                             x = semi1*rad*cos(angle);
231                                             y = semi2*rad*sin(angle);
232                                     }else{
233                                             x=2*distx*drand48()-distx;
234                                             y=2*disty*drand48()-disty;
235                                             cout<<y<<endl;
236                                     }
237                             }
238                             else{
```

```
239                                     x=0.0; //Eller x=drand48()*vx
240                                     y=0.0;
241                             }
242                             if(gauss==1){
243                                     double Ran1, Ran2, Ran3;
244                                     Ran1 = drand48();
245                                     Ran2 = drand48();
246                                     cout<<Ran1<<" "<<Ran2<<endl;
247                                     Ran3 = sqrt(-2*log(Ran1))*cos(2*pi*Ran2);
248                                     Q = 1.0/2.0*Ran3;
249                                     //cout<<Q<<endl;
250                                     while(Q==0.0){
251                                             Ran1 = drand48();
252                                             Ran2 = drand48();
253                                             cout<<"Inside loop:"<<Ran1<<" "<<Ran2<<endl;
254                                             Ran3 = sqrt(-2*log(Ran1))*cos(2*pi*Ran2);
255                                             cout<<"Inside loop Ran3=:"<<Ran3<<endl;
256                                             Q = 1.0/2.0*Ran3;
257                                             //cout<<Q<<endl;
258                                     }
259                             }else if(gauss==0){
260                                     Q=2.0*drand48()-1.0;
261                                     while(Q==0.0)Q=drand48();
262                                     if(Q<0.0)Q = -1.0;
263                                     else if(Q>0.0) Q = 1.0;
264                                     else cout<<"DAFUQ"<<" Q="<<Q<<endl;
265                                     Q_eff += Q;
266 //                                  cout<<"Q_eff="<<Q_eff<<endl;
267                                     #ifdef BALANCED
268                                     if(i<Nsgen/2) Q = 1.0;
269                                     else Q = -1.0;
270                                     Q_eff2 += Q;
271 //                                  cout<<"Q_eff2="<<Q_eff2<<endl;
272                                     #endif
273                             }else if(gauss==2){
274                                     Q=2.0*drand48()-1.0;
275                                     while(Q==0.0)Q=2.0*drand48()-1.0;
276                             }else cout<<"gauss="<<gauss<<endl;
277                             Q /= 10.0;
278                             s[Ns++].setall(x,y,Q,B);
279                     }
280
281             }else cout<<"could not generate"<<endl;
282             if(Npgen+Np<=Npmax){
283                     for(int i=0; i<Npgen; ++i){
284                             x=drand48();
285                             y=drand48();
286                             u=drand48();
287                             v=drand48();
288                             p[Np++].setall(x,y,u,v,q,m);
289                     }
290
291 #endif
292             }cout<<"Ns="<<Ns<<" "<<"Np="<<Np<<endl;
293 #ifdef TEST
294             for(int i=0; i<Ntgen;++i){
295                     x = 2*distx*drand48()-distx;
296                     y = 2*disty*drand48()-disty;
297                     if(gauss==1){
298                             double Ran1, Ran2, Ran3;
299                             Ran1 = drand48();
300                             Ran2 = drand48();
301                             cout<<Ran1<<" "<<Ran2<<endl;
302                             Ran3 = sqrt(-2*log(Ran1))*cos(2*pi*Ran2);
303                             Q = 1.0/2.0*Ran3;
304                             //cout<<Q<<endl;
305                             while(Q==0.0){
306                                     Ran1 = drand48();
307                                     Ran2 = drand48();
308                                     cout<<"Inside loop:"<<Ran1<<" "<<Ran2<<endl;
309                                     Ran3 = sqrt(-2*log(Ran1))*cos(2*pi*Ran2);
310                                     cout<<"Inside loop Ran3=:"<<Ran3<<endl;
311                                     Q = 1.0/2.0*Ran3;
312                                     //cout<<Q<<endl;
313                             }
314                     }else if(gauss==0){
315                             Q=2.0*drand48()-1.0;
316                             while(Q==0.0)Q=drand48();
317                             if(Q<0.0)Q = -1.0;
318                             else if(Q>0.0) Q = 1.0;
319                             else cout<<"DAFUQ"<<" Q="<<Q<<endl;
320                             //cout<<"Q="<<Q<<endl;
321                     }else if(gauss==2){
```

```
322                         Q=2.0*drand48() -1.0;
323                         while(Q==0.0)Q=2.0*drand48() -1.0;
324                 }else cout<<"gauss="<<gauss<<endl;
325                 test[Nt++].setall(x,y,Q,B);
326
327         }
328
329 #endif
330 //      cout<<"Q_eff="<<Q_eff<<" Q_eff2="<<Q_eff2<<endl;
331 //      cin>>Q_eff;
332 }
333
334
335 //Calculates the force on a paricle in the field generated by all other strings and
        particles
336 void moveparticles(int Mmax, int Nmax){
337         double Fqx, Fqy,x,y,u,v;
338         double Ex=0.0, Ey=0.0, B=0.0, Etx, Ety, Bt;
339         int nbox, mbox;
340         // cout<<"moveparticles"<<endl;
341         for(int i=0; i<Np; ++i){
342                 Ex=Ey=B=0.0;
343                 Etx=Ety=Bt=0.0;
344                 //cout<<"main::moveparticles: i="<<i<< endl;
345                 Etx=Ety=Bt=0.0; //Temp variables
346                 p[i].getposvel(&x,&y,&u,&v);
347                 for(mbox=-Mmax;mbox<=Mmax;++mbox){
348                         for(nbox=-Nmax;nbox<=Nmax;++nbox){
349                                 for(int j=0; j<i; ++j){
350                                         //cout<<"moveparticles(lower) j="<<j<<endl;
351                                         p[j].getE(x,y,&Etx,&Ety,mbox,nbox,L);
352                                         Ex+=Etx, Ey+=Ety;
353                                         //cout<<"lower: Ex="<<Ex<<" Ey="<<Ey<<endl;
354                                 }
355                                 for(int j=i+1; j<Np; ++j){
356                                         //cout<<"moveparticles(higher) j="<<j<<endl;
357                                         p[j].getE(x,y,&Etx,&Ety,mbox,nbox,L);
358                                         Ex+=Etx, Ey+=Ety;
359                                         //cout<<"higher: Ex="<<Ex<<" Ey="<<Ey<<endl;
360                                 }
361                                 for(int j=0; j<Ns; ++j){
362                                         //cout<<"moveparticles(strings) j="<<j<<endl;
363                                         s[j].getEB(x,y,&Etx,&Ety,&Bt,mbox,nbox,L, &
                                                placeholder);
364                                         Ex+=Etx, Ey+=Ety, B+=Bt;
365                                         //cout<<"strings: Ex="<<Ex<<" Ey="<<Ey<<" B="<<B
                                                <<endl;
366                                 }
367                         }
368                 }
369                 B = B/Ns;
370                 //cout<<"B="<<B<<endl;
371                 //Calculate F/q
372                 Fqx = K*Ex + v*B;
373                 Fqy = K*Ey - u*B;
374
375                 //Pass force/charge to particle i
376                 p[i].setFq(Fqx,Fqy);
377                 //cout<<endl;
378         }
379 }
380
381 //Calculates the ExB-velocity for a string in the field form all other stings and
        particles
382 void movestrings(int Mmax, int Nmax, double t){
383         leastdist = 1.0e30;
384         double distance = 0.0;
385         double x,y,u,v;
386         double Ex=0.0, Ey=0.0, B=0.0, Etx, Ety, Bt;
387         int mbox,nbox;
388         double vabs2,vmax2 = 0.0, fac = pi/10.0;
389
390         // cout<<"movestrings"<<endl;
391         #pragma omp parallel for num_threads(1)
392         for(int i=0; i<Ns; ++i){
393                 Ex=Ey=B=0.0;
394                 Etx=Ety=Bt=0.0; //Temp variables
395                 s[i].getpos(&x,&y);
396                 for(mbox=-Mmax;mbox<=Mmax;++mbox){
397                         //cout<<"mbox="<<mbox<<endl;
398                         for(nbox=-Nmax;nbox<=Nmax;++nbox){
399                                 //cout<<"nbox="<<nbox<<endl;
400                                 Etx=Ety=Bt=0.0; //Temp variables
```

```
401
402                                     for(int j=0; j<Np; ++j){
403                                             // cout<<"movestrings: x="<<x<<" y="<<y<<endl;
404                                             p[j].getE(x,y,&Etx,&Ety,mbox,nbox, L);
405                                             Ex+=Etx, Ey+=Ety;
406                                     }
407
408                                     for(int j=0; j<Ns; ++j){
409                                             //cout << "j="<<j<<endl;
410                                             s[j].getEB(x,y,&Etx,&Ety,&Bt,mbox,nbox, L, &
                                                    distance);
411                                             Ex+=Etx, Ey+=Ety;
412                                             B+=Bt;
413                                             if(i!=j){
414                                                     if(distance<leastdist){
415                                                             leastdist=distance;
416                                                     }
417                                             }
418                                     }
419                             }
420                     }
421                     /*      for(int j=0; j<i ; ++j){
422                     s[j].getEB(x,y,&Etx,&Ety,&Bt);
423                     Ex+=Etx, Ey+=Ety, B+=Bt;
424                     }
425
426                     for(int j=i+1; j<Ns; ++j){
427                     s[j].getEB(x,y,&Etx,&Ety,&Bt);
428                     Ex+=Etx, Ey+=Ety, B+=Bt;
429                     }
430                     */
431                     //B = B/(Ns-1.0);
432                     B /= (Ns*((Mmax*2+1)*(Nmax*2+1)));
433                     //cout<<"movestrings: i="<<i<<" B="<<B<<endl;
434                     //Calculate ExB velocity
435                     if(B!=0) u = 1.0*K*Ey/B, v = -1.0*K*Ex/B;
436                     else u=v=0.0;
437
438                     vabs2 = v*v+u*u;
439                     if (vabs2 > vmax2) vmax2 = vabs2;
440                     //Pass force/charge to particle i
441                     s[i].setv(u,v);
442             }
443     #ifdef TEST
444             for(int i=0; i<Nt; ++i){
445                     Ex=Ey=B=0.0;
446                     Etx=Ety=Bt=0.0; //Temp variables
447                     test[i].getpos(&x,&y);
448                     for(mbox=-Mmax;mbox<=Mmax;++mbox){
449                             //cout<<"mbox="<<mbox<<endl;
450                             for(nbox=-Nmax;nbox<=Nmax;++nbox){
451                                     //cout<<"nbox="<<nbox<<endl;
452                                     Etx=Ety=Bt=0.0; //Temp variables
453
454                                     for(int j=0; j<Np; ++j){
455                                             // cout<<"movestrings: x="<<x<<" y="<<y<<endl;
456                                             p[j].getE(x,y,&Etx,&Ety,mbox,nbox, L);
457                                             Ex+=Etx, Ey+=Ety;
458                                     }
459
460                                     for(int j=0; j<Ns; ++j){
461                                             //cout << "j="<<j<<endl;
462                                             s[j].getEB(x,y,&Etx,&Ety,&Bt,mbox,nbox, L, &
                                                    distance);
463                                             Ex+=Etx, Ey+=Ety;
464                                             B+=Bt;
465                                             if(i!=j){
466                                                     if(distance<leastdist){
467                                                             leastdist=distance;
468                                                     }
469                                             }
470                                     }
471                             }
472                     }
473                     /*      for(int j=0; j<i ; ++j){
474                     s[j].getEB(x,y,&Etx,&Ety,&Bt);
475                     Ex+=Etx, Ey+=Ety, B+=Bt;
476                     }
477
478                     for(int j=i+1; j<Ns; ++j){
479                     s[j].getEB(x,y,&Etx,&Ety,&Bt);
480                     Ex+=Etx, Ey+=Ety, B+=Bt;
481                     }
```

```
482                     */
483                     //B = B/(Ns-1.0);
484                     B /= (Ns*((Mmax*2+1)*(Nmax*2+1)));
485     //              cout<<"movestrings B="<<B<<endl;
486     //              cin>>holder;
487                     //cout<<"movestrings: i="<<i<<" B="<<B<<endl;
488                     //Calculate ExB velocity
489                     if(B!=0) u = 1.0*K*Ey/B, v = -1.0*K*Ex/B;
490                     else u=v=0.0;
491
492                     vabs2 = v*v+u*u;
493                     if (vabs2 > vmax2) vmax2 = vabs2;
494                     //Pass force/charge to particle i
495                     test[i].setv(u,v);
496             }
497     #endif
498
499             double dt_max = 0.1*(0.5*((x_max-x_min) + (y_max-y_min))/sqrt(vmax2));
500             l_max = fac*leastdist;
501     //      if(vmax2 > 0.0 && l_max/sqrt(vmax2) < dt){
502                     cout<<"Endrer dt fra:"<<dt;
503                     dt = fac * l_max/sqrt(vmax2);
504     //              if(dt<1.0e-7)dt=1.0e-8;
505     //              else if(dt<1.0e-6)dt=1.0e-7;
506     //              else if(dt<1.0e-5)dt=1.0e-6;
507     //              else if(dt<1.0e-4)dt=1.0e-5;
508     //              else if(dt<1.0e-3)dt=1.0e-4;
509     //              else if(dt<1.0e-2)dt=1.0e-3;
510     //              else if(dt<1.0e-1)dt=1.0e-2;
511     //              else if(dt<1.0e-0)dt=1.0e-1;
512     //              else dt=1.0e-1;
513     /*              if(t+dt >= nextstep){
514                             dt = dt - (t + dt - nextstep);
515                             nextstep += interval;
516                             //cout <<"nextstep="<<nextstep<<endl;
517                             printtest = true;
518                     }*/
519                     cout<<" til:"<<dt<<" dt_max"<<dt_max<<endl;
520                     if(dt_max > dt){
521                             for(int i=0; i<Ns; ++i){
522                                     s[i].setdt(dt);
523                             }
524                             #ifdef TEST
525                             for(int i=0;i<Nt;++i){
526                                     test[i].setdt(dt);
527                             }
528                             #endif
529                     }
530
531     //      }
532     }
533
534     void setiterationlength(){
535             dt=(pi*leastdist*leastdist)/(10.0);
536             l_max=dt/leastdist;
537     //      for(int i=0; i<Ns; ++i){
538     //              s[i].setdt(dt);
539     //      }
540     }
541
542     void E(double x, double y, double &Ex, double &Ey){
543       Ex=0, Ey=0;
544       double Etx=0, Ety=0, Bt=0.0; //Temp variables
545       for(int j=0; j<Np; ++j){
546         p[j].getE(x,y,&Etx,&Ety, 0, 0, 0.0);
547         Ex+=Etx, Ey+=Ety;
548       }
549       for(int j=0; j<Ns ; ++j){
550         s[j].getEB(x,y,&Etx,&Ety,&Bt, 0, 0, 0.0, &placeholder);
551         Ex+=Etx, Ey+=Ety;
552       }
553       //Regn ut kraften fra likningen i movestrings, og movepart
554
555
556     }
557
558     //Calculates ExB
559     void ExB(double x, double y, double &ExBx, double &ExBy){
560             double Ex=0, Ey=0, B=0;
561             double Etx=0, Ety=0, Bt=0.0; //Temp variables
562             int mbox, nbox;
563             for(mbox=-Mmax;mbox<=Mmax;++mbox){
564                     //cout<<"mbox="<<mbox<<endl;
```

```
565                     for(nbox=-Nmax;nbox<=Nmax;++nbox){
566                             for(int j=0; j<Np; ++j){
567                                     p[j].getE(x,y,&Etx,&Ety, mbox, nbox, L);
568                                     Ex+=Etx, Ey+=Ety;
569                             }
570                             for(int j=0; j<Ns ; ++j){
571                                     s[j].getEB(x,y,&Etx,&Ety,&Bt, mbox, nbox, L, &
                                            placeholder);
572                                     Ex+=Etx, Ey+=Ety, B+=Bt;
573                             }
574                     }
575             }
576       //Regn ut ExB fra likningen i movestrings, og movepart
577       //returneres som pekere, gitt i argument
578
579
580             B /= (Ns*((Mmax*2+1)*(Nmax*2+1)));
581     //      cout<<"ExB B="<<B;
582     //      cin>>holder;
583             ExBx = 1.0*K*Ey/B;
584             ExBy = -1.0*K*Ex/B;
585     }
586
587     void hamiltonian(double &hamilton){
588             double x,y,q,b;
589             double pottmp;
590             int mbox, nbox;
591             hamilton = 0.0;
592             for(int i=0; i<Ns; ++i){
593
594                     pottmp=0.0; //Temp variables
595                     s[i].getall(&x,&y,&q,&b);
596                     for(mbox=-Mmax;mbox<=Mmax;++mbox){
597                             for(nbox=-Nmax;nbox<=Nmax;++nbox){
598                                     pottmp = 0.0; //Temp variables
599                                     for(int j=0; j<Ns; ++j){
600                                             s[j].getpotential(x,y,&pottmp,mbox,nbox, L,q);
601                                             hamilton += pottmp;
602     //                                      cout<<"main_hamiltonian"<<hamilton<<endl;
603                                             if(hamilton>1e30)exit(1);
604                                     }
605                             }
606                     }
607             }
608     //      hamilton = hamilton*(-1/(4*pi))
609     }
610     //Generates blank meshgrid files
611     void meshclear(){
612             ofstream meshE;
613             ofstream meshEB;
614             meshE.open("data/mesh/meshElist.m", ios::trunc);
615             meshEB.open("data/mesh/meshEBlist.m",ios::trunc);
616             meshE<<"function"<<" "<<"meshElist"<<endl;
617             meshEB<<"function"<<" "<<"meshEBlist"<<endl;
618             meshE.close();
619             meshEB.close();
620     }
621
622     //Calculates, and dumps to file the E-fild and ExB-fields at the point in a meshgrid
623     void meshforce(int timestep){
624
625             stringstream strtime;
626             strtime<<timestep;
627
628
629
630             ofstream Exout;
631             ofstream Eyout;
632             ofstream EBxout;
633             ofstream EByout;
634             ofstream Xout;
635             ofstream Yout;
636             ofstream meshE;
637             ofstream meshEB;
638
639             if(timestep >= 0 && timestep <=9){
640                     Exout.open(("data/mesh/Efieldx0000"+strtime.str()+".dat").c_str(),ios::
                            trunc);
641                     Eyout.open(("data/mesh/Efieldy0000"+strtime.str()+".dat").c_str(),ios::
                            trunc);
642                     EBxout.open(("data/mesh/ExBx0000"+strtime.str()+".dat").c_str(),ios::
                            trunc);
```

```
643                 EByout.open(("data/mesh/ExBy0000"+strtime.str()+".dat").c_str(),ios::
                        trunc);
644         }
645         else if(timestep > 9 && timestep <= 99){
646                 Exout.open(("data/mesh/Efieldx000"+strtime.str()+".dat").c_str(),ios::
                        trunc);
647                 Eyout.open(("data/mesh/Efieldy000"+strtime.str()+".dat").c_str(),ios::
                        trunc);
648                 EBxout.open(("data/mesh/ExBx000"+strtime.str()+".dat").c_str(),ios::
                        trunc);
649                 EByout.open(("data/mesh/ExBy000"+strtime.str()+".dat").c_str(),ios::
                        trunc);
650         }
651         else if(timestep > 99 && timestep <= 999){
652                 Exout.open(("data/mesh/Efieldx00"+strtime.str()+".dat").c_str(),ios::
                        trunc);
653                 Eyout.open(("data/mesh/Efieldy00"+strtime.str()+".dat").c_str(),ios::
                        trunc);
654                 EBxout.open(("data/mesh/ExBx00"+strtime.str()+".dat").c_str(),ios::trunc
                        );
655                 EByout.open(("data/mesh/ExBy00"+strtime.str()+".dat").c_str(),ios::trunc
                        );
656         }
657         else if(timestep > 999 && timestep <= 9999){
658                 Exout.open(("data/mesh/Efieldx0"+strtime.str()+".dat").c_str(),ios::
                        trunc);
659                 Eyout.open(("data/mesh/Efieldy0"+strtime.str()+".dat").c_str(),ios::
                        trunc);
660                 EBxout.open(("data/mesh/ExBx0"+strtime.str()+".dat").c_str(),ios::trunc)
                        ;
661                 EByout.open(("data/mesh/ExBy0"+strtime.str()+".dat").c_str(),ios::trunc)
                        ;
662         }
663         else if(timestep > 9999){
664                 Exout.open(("data/mesh/Efieldx"+strtime.str()+".dat").c_str(),ios::trunc
                        );
665                 Eyout.open(("data/mesh/Efieldy"+strtime.str()+".dat").c_str(),ios::trunc
                        );
666                 EBxout.open(("data/mesh/ExBx"+strtime.str()+".dat").c_str(),ios::trunc);
667                 EByout.open(("data/mesh/ExBy"+strtime.str()+".dat").c_str(),ios::trunc);
668         }
669         else cout << "meshforce: timestep not in valid range" << timestep << endl;
670
671         meshE.open("data/mesh/meshElist.m", ios::app);
672         meshEB.open("data/mesh/meshEBlist.m", ios::app);
673
674         if(timestep == 0){
675                 Xout.open("data/mesh/Xmesh.dat");
676                 Yout.open("data/mesh/Ymesh.dat");
677                 meshE<<"load('Xmesh.dat')"<<endl;
678                 meshE<<"load('Ymesh.dat')"<<endl;
679                 meshEB<<"load('Xmesh.dat')"<<endl;
680                 meshEB<<"load('Ymesh.dat')"<<endl;
681         }
682
683
684         meshE<<"load('"<<"Efieldx"<<strtime.str()<<".dat')"<<endl;
685         meshE<<"load('"<<"Efieldy"<<strtime.str()<<".dat')"<<endl;
686         meshEB<<"load('"<<"ExBx"<<strtime.str()<<".dat')"<<endl;
687         meshEB<<"load('"<<"ExBy"<<strtime.str()<<".dat')"<<endl;
688
689         int nx = 12, ny = 12;
690         double x0 = xl, x1 = xu, dx = (x1-x0)/(nx-1);
691         double y0 = yl, y1 = yu, dy = (y1-y0)/(ny-1);
692         double Ex,Ey;
693         double ExBx,ExBy;
694         for(double y=y0; y<=y1; y+=dy){
695                 for(double x=x0; x<=x1; x+=dx){
696                         E(x,y,Ex,Ey);
697                         ExB(x,y,ExBx,ExBy);
698                         if(timestep == 0){
699                                 Yout<<y<<" ";
700                                 Xout<<x<<" ";
701                         }
702                         Exout<<Ex<<" ";
703                         Eyout<<Ey<<" ";
704                         EBxout<<ExBx<<" ";
705                         EByout<<ExBy<<" ";
706                 }
707                 if(timestep == 0){
708                         Yout<<endl;
709                         Xout<<endl;
710                 }
```

```
711                     Exout<<endl;
712                     Eyout<<endl;
713                     EBxout<<endl;
714                     EByout<<endl;
715             }
716             Yout.close();
717             Xout.close();
718             Exout.close();
719             Eyout.close();
720             EBxout.close();
721             EByout.close();
722             meshE.close();
723             meshEB.close();
724     }
725
726
727     //Actually steps all the particles and strings forwarg one timestep. Eventual boundary
            conditions are apply here
728     void integrate(){
729             double x_s, x_p, y_s, y_p, u_p, v_p, r_p, r_s, q_p, m_p, q_s, b_s, x_test,
                    y_test;
730             double *X = new double [Ns];
731             double *Y = new double [Ns];
732             double *Vx = new double [Ns];
733             double *Vy = new double [Ns];
734             double x_t,y_t,vx_t,vy_t;
735
736             #pragma omp parallel for
737             for(int i=0; i<Np; ++i){
738                     p[i].integrate();
739     #ifdef BOUNDARY
740                     p[i].getposvel(&x_p,&y_p,&u_p,&v_p);
741                     r_p = sqrt(x_p*x_p+y_p*y_p);
742                     while(r_p > Rmaks){
743                             //TODO Does not work, need setall or something also need getall(
                                    x,y,u,v,Q,m)
744                             p[Np-1].getall(&x_p,&y_p,&u_p,&v_p,&q_p,&m_p);
745                             p[i].setall(x_p,y_p,u_p,v_p,q_p,m_p);
746                             Np=Np-1;
747                             r_p = sqrt(x_p*x_p+y_p*y_p);
748                     }
749     #endif
750     #ifdef PERIODIC
751                     p[i].getposvel(&x_p,&y_p,&u_p,&v_p);
752                     if(x_p>x_max) x_p -= (x_max-x_min);
753                     if(x_p<x_min) x_p += (x_max-x_min);
754                     if(y_p>y_max) y_p -= (y_max-y_min);
755                     if(y_p<y_min){ cout<<"y_p="<<y_p<<endl; y_p += (y_max-y_min); cout<<"new
                            y_p="<<y_p<<endl;}
756                     p[i].setpos(x_p,y_p);
757
758     #endif
759             }
760
761
762             for(int i=0;i<Ns;++i){
763                     s[i].getpos(&x_t,&y_t);
764                     s[i].getv(&vy_t,&vy_t);
765                     X[i]=x_t; Y[i]=y_t; Vx[i]=vx_t; Vy[i]=vy_t;
766
767
768              }
769
770             for(int i=0; i<Ns; ++i){
771                     s[i].integrate();
772             }
773
774
775     //      bool ok=true;
776     //      while(ok){
777     //      #pragma omp parallel for
778     //      ok = false;
779     //              for(int i=0; i<Ns; ++i){
780     //                      s[i].integrate();
781     //                      double lx, ly, l_t;
782     //                      double ratio;
783     //
784     //                      s[i].getpos(&x_s,&y_s);
785     //                      lx=x_s-X[i]; ly=y_s-Y[i];
786     //                      l_t=sqrt(lx*lx+ly*ly);
787     //                      if(l_t>l_max){
788     //                              //cout<<"dt1="<<dt;
789     //                              ratio=(l_t/l_max);
```

```
790 //                                      cout<<"ratio="<<ratio<<endl;
791 //                                      dt = 1.0/ratio*dt;
792 //                                      //cout<<"dt2="<<dt<<endl;
793 //                                      ok = true;
794 //                                      for(int i=0;i<Ns;++i){
795 //                                              s[i].setpos(X[i],Y[i]);
796 //                                              s[i].setv(Vx[i],Vy[i]);
797 //                                              s[i].setdt(dt);
798 //                                      }
799 //                                      break;
800 //                              }
801 //
802 //                      }
803 //      }
804
805
806 #ifdef BOUNDARY
807                       for(int i=0;i<Ns;++i){
808                               s[i].getpos(&x_s,&y_s);
809                               r_s = sqrt(x_s*x_s+y_s*y_s);
810                               while(r_s > Rmaks){
811                                       s[Ns-1].getall(&x_s,&y_s,&q_s,&b_s);
812                                       s[i].setall(x_s,y_s,q_s,b_s);
813                                       Ns=Ns-1;
814                                       r_s = sqrt(x_s*x_s+y_s*y_s);
815                               }
816                       }
817 #endif
818 #ifdef PERIODIC
819                       for(int i=0;i<Ns;++i){
820                               s[i].getpos(&x_s,&y_s);
821                               if(x_s>x_max) x_s -= (x_max-x_min);
822                               if(x_s<x_min) x_s += (x_max-x_min);
823                               if(y_s>y_max) y_s -= (y_max-y_min);
824                               if(y_s<y_min) y_s += (y_max-y_min);
825                               s[i].setpos(x_s, y_s);
826                       }
827 #endif
828
829 #ifdef TEST
830           for(int i=0;i<Nt;++i){
831                   test[i].integrate();
832           }
833 #ifdef PERIODIC
834           for(int i=0;i<Nt;++i){
835                   test[i].getpos(&x_test, &y_test);
836                   if(x_test>x_max) x_test -= (x_max-x_min);
837                   if(x_test<x_min) x_test += (x_max-x_min);
838                   if(y_test>y_max) y_test -= (y_max-y_min);
839                   if(y_test<y_min) y_test += (y_max-y_min);
840                   test[i].setpos(x_test,y_test);
841           }
842 #endif
843 #endif
844
845 #ifdef BOUNDARY
846           //influx-rutine:
847           double Ps_flux=0.1;
848           int input=int(Ps_flux);
849           double frac=Ps_flux-input;
850           if(input>0) generate(0,input,false);
851           if(drand48()<frac) generate(0,1,false);
852 #endif
853
854 }
855
856 //Declare number of timesteps and maximum time, sets dt
857 void declare(double &T, double &dt, int &n){
858           cout << "Tmax=" << endl;
859           cin >> T;
860           //cout << "n=" << endl;
861           //cin >> n;
862
863           double density=(Nsmax+Npmax)/(distx*disty);
864           double length = sqrt(1/density);
865           double V0 = 1*1/(length*1);
866           double dr=pi*length/100.0;//TODO en konstant
867           //a=TODO en konstant
868           dt=dr/(V0);
869 //        if(dt<1.0e-7)dt=1.0e-8;
870 //        else if(dt<1.0e-6)dt=1.0e-7;
871 //        else if(dt<1.0e-5)dt=1.0e-6;
872 //        else if(dt<1.0e-4)dt=1.0e-5;
```

```
873 //        else if(dt<1.0e-3)dt=1.0e-4;
874 //        else if(dt<1.0e-2)dt=1.0e-3;
875 //        else if(dt<1.0e-1)dt=1.0e-2;
876 //        else if(dt<1.0e-0)dt=1.0e-1;
877 //        else dt=1.0e-1;
878           l_max = 10*dr;
879           cout<<"declare: dt="<<dt<<endl;
880           //exit(1);
881           cout<<"Q-distribution (1=gaussian, 0=discrete, 2=uniform):"<<endl;
882           cin >> gauss;
883           //dt = T/((double)n);
884 #ifdef PERIODIC
885           cout << "Mmax=" << endl;
886           cin >> Mmax;
887           cout << "Nmax=" << endl;
888           cin >> Nmax;
889 #else
890           Mmax = 0;
891           Nmax = 0;
892 #endif
893 #ifdef FOURCLOUD
894           cout<<"define impact factor"<<endl;
895           cin>>impact;
896 #endif
897 }
898
899 //Initializes the file middel.dat
900 void initmiddel(){
901           ofstream utmiddel;
902           utmiddel.open("data/middel.dat", ios::trunc);
903           utmiddel.close();
904 }
905
906 //Calculates the mean value and the mean square value of the strings at the time of
        calling
907 void middel(double t, int step){
908           double x, y, r, r2, rm=0.0, r2m = 0.0;
909           for(int i=0; i<Ns; ++i){
910                     s[i].getpos(&x,&y);
911                     r2 = x*x+y*y;
912                     r = sqrt(r2);
913                     rm += r;
914                     r2m += r2;
915           }
916
917           //cout<<r<<" "<<r2<<endl;
918
919           r2m = r2m/Ns;
920           rm = rm/Ns;
921
922           ofstream utmiddel;
923           utmiddel.open("data/middel.dat", ios::app);
924           utmiddel << t << " " << step << " " << rm << " " << r2m << endl;
925           utmiddel.close();
926 }
927
928 void vmean(int timestep, int Mmax){
929           int xi,yi,number;
930           number = 7;
931           cout<<"Ns="<<Ns<<endl;;
932           stringstream strtime;
933           stringstream strM;
934           strtime<<timestep;
935           strM<<Mmax;
936 //        cout<<"BLOCK 1"<<endl;
937
938           ofstream velocitymeanradial;
939           ofstream velocitymeantheta;
940 //        cout<<"BLOCK 2"<<endl;
941
942           velocitymeanradial.open(("data/velocitymeanradialM"+strM.str()+"t"+strtime.str()
                +".dat").c_str(),ios::trunc);
943           velocitymeantheta.open(("data/velocitymeanthetaM"+strM.str()+"t"+strtime.str()+"
                .dat").c_str(),ios::trunc);
944 //        cout<<"BLOCK 3"<<endl;
945
946           double x,y,ExBx,ExBy,r,theta,dl,vx,vy,xit,yit, V, sinphi, phi;
947
948 //        cout<<"number="<<number<<endl;
949           dl=2*distx/number;
950 //        cout<<"dl="<<dl<<endl;
951           double **vmeanr=new double*[number];
952           double **vmeanth=new double*[number];
```

```
953         double **counter=new double*[number];
954 //      cout<<"BLOCK 4"<<endl;
955
956         for(int i=0;i<number;++i){
957                 vmeanr[i]=new double[number];
958                 vmeanth[i]=new double[number];
959                 counter[i]=new double[number];
960 //              cout<<i;
961         }
962 //      cout<<"BLOCK 5"<<endl;
963
964         for(int i=0;i<number;++i){
965 //              cout<<i<<" ";
966                 for(int j=0;j<number;++j){
967                         vmeanr[i][j] = 0.0;
968                         vmeanth[i][j] = 0.0;
969                         counter[i][j] = 0.0;
970 //                      cout<<j<<" ";
971                 }
972 //              cout<<endl;
973         }
974 //      cout<<"BLOCK 6"<<endl;
975
976 //Calculates the velocity histogram
977         for(int i=0; i<Ns; ++i){
978                 cout<<i<<" ";
979                 s[i].getpos(&x,&y);
980                 r=sqrt(x*x+y*y);
981
982                 //cout<<"x="<<x<<" y="<<y<<endl;
983                 //cout<<"x/dl="<<x/dl<<" y/dl="<<y/dl<<endl;
984                 xi=(x+distx)/dl; yi=(y+disty)/dl;
985                 ExB(x,y,ExBx,ExBy);
986                 vx=ExBx; vy=ExBy;
987                 //s[i].getv(&vx,&vy);
988                 cout<<" vx="<<vx<<" vy="<<vy<<endl;
989                 V = sqrt(vx*vx+vy*vy);
990                 //cout<<"xi="<<xi<<" "<<"yi="<<yi<<endl;
991                 sinphi=(x*vy-y*vx)/(r*V);
992                 phi = asin(sinphi);
993                 vmeanr[xi][yi]+=V*cos(phi);
994                 vmeanth[xi][yi]+=V*sin(phi);
995                 counter[xi][yi]++;
996
997         }
998         cout<<"BLOCK 7"<<endl;
999         for(int i=0;i<number;++i){
1000                for(int j=0;j<number;++j){
1001                        if(counter[i][j]!=0){
1002                                vmeanr[i][j]/=counter[i][j];
1003                                vmeanth[i][j]/=counter[i][j];
1004                        }
1005                }
1006        }
1007        cout<<"BLOCK 8"<<endl;
1008        for(int i=0;i<number;++i){
1009                for(int j=0;j<number;++j){
1010                  velocitymeanradial<<vmeanr[i][j]<<" ";
1011                  velocitymeantheta<<vmeanth[i][j]<<" ";
1012                }
1013                velocitymeanradial<<endl;
1014                velocitymeantheta<<endl;
1015        }
1016        cout<<"BLOCK 9"<<endl;
1017        velocitymeantheta.close();
1018        velocitymeanradial.close();
1019        cout<<"BLOCK 10"<<endl;
1020 }
1021
1022 void corner(int Mmax){
1023        ofstream corners;
1024        corners.open("data/corners.dat",ios::app);
1025
1026        double x,y,ExBx,ExBy,vx,vy, V;
1027        x=x_max; y=y_max;
1028        ExB(x,y,ExBx,ExBy);
1029        vx=ExBx; vy=ExBy; V=sqrt(vx*vx+vy*vy);
1030
1031        corners<<Mmax<<" "<<V<<" "<<vx<<" "<<vy<<endl;
1032        corners.close();
1033 }
1034
1035
```

```
1036 //Initializes the histogram files
1037 void inithist(){
1038         int ix, iy;
1039         double x, y;
1040         double Lx=(xu-xl);
1041         double Ly=(yu-yl);
1042         double xbin=(Lx)/(XBINS);
1043         double ybin=(Ly)/(YBINS);
1044         double dxdy=xbin*ybin;
1045         ofstream histmeshx, histmeshy;
1046         ofstream histm;
1047         histm.open("data/hist/histlist.m", ios::trunc);
1048         histmeshx.open("data/hist/histmeshx.dat",ios::trunc);
1049         histmeshy.open("data/hist/histmeshy.dat",ios::trunc);
1050         for(ix=0,x=xl ; ix<=XBINS ; ++ix,x+=xbin){
1051                 for(iy=0,y=yl ; iy<=YBINS ; ++iy,y+=ybin){
1052                         histmeshx<<x<<" ";
1053                         histmeshy<<y<<" ";
1054                 }
1055                 histmeshx<<endl;
1056                 histmeshy<<endl;
1057         }
1058
1059         histm<<"function"<<" "<<"histlist"<<endl;
1060         histm<<"load('histmeshx.dat')"<<endl;
1061         histm<<"load('histmeshy.dat')"<<endl;
1062
1063         histmeshx.close();
1064         histmeshy.close();
1065         histm.close();
1066
1067 }
1068
1069 //Generates the histograms
1070 void histogram(int timestep){
1071
1072         string str,name;
1073         double x,y,ybin,xbin;
1074         int ix,iy;
1075
1076         //Lx = lengden av boks i x-retn.
1077         //Ly = lengden av boks i x-retn.
1078         //xu,yu = Upper limit for x,y
1079         //xl,yl = nedre grense for x,y
1080
1081         double Lx=(xu-xl);
1082         double Ly=(yu-yl);
1083         xbin=(Lx)/(XBINS);
1084         ybin=(Ly)/(YBINS);
1085         double dxdy=xbin*ybin;
1086         cout << "xbin="<<xbin<<" ybin="<<ybin<<endl;
1087
1088         double hist[XBINS+1][YBINS+1];
1089         int errorcount=0, histcount=0;
1090
1091
1092 #if HIST == 1
1093 //NB!!!: X,Y-mesh displced before use
1094         //Alternative 1: The easy way. Not used
1095         for(int i=0; i<=XBINS; ++i)for(int j=0;j<=YBINS; ++j)hist[i][j]=0.0;
1096         for(int j=0; j<Ns; ++j){
1097                 s[j].getpos(&x,&y);
1098                 ix=int((x-xl)/xbin); //array posisjon i x
1099                 iy=int((y-yl)/ybin); //array posisjon i y
1100                 if(ix>=0 && ix<=XBINS && iy>=0 && iy<=YBINS){
1101                         hist[ix][iy]++;
1102                         histcount++;
1103                 }else errorcount++;
1104         }
1105 #endif
1106
1107 #if HIST == 2
1108
1109         double wx,wy;
1110         //Alternative 2: More complicated. Used
1111         for(int i=0; i<=XBINS; ++i)for(int j=0;j<=YBINS; ++j)hist[i][j]=0.0;
1112         for(int j=0; j<Ns; ++j){
1113                 //Finn posisjon av gridpunkt "under" x,y
1114                 //wx,wy midlertidig variabel
1115                 s[j].getpos(&x,&y);
1116                 ix = int(wx=((x-xl)/xbin));
1117                 iy = int(wy=((y-yl)/ybin));
1118                 wx-=(double)ix;
```

```
1119                    wy-=(double)iy;
1120                    if(ix>=0 && ix<=XBINS && iy>=0 && iy<=YBINS){
1121                            hist[ix][iy]+=(1.0-wx)*(1.0-wy);
1122                            hist[ix+1][iy]+=wx*(1.0-wy);
1123                            hist[ix][iy+1]+=(1.0-wx)*(wy);
1124                            hist[ix+1][iy+1]+=wx*(wy);
1125                            histcount++;
1126                    }else{
1127                            errorcount++;
1128                            //cout<<"hist alt.2: wrong ix,iy="<<ix<<","<<iy<<" wx,wy="<<wx
                                    <<","<<wy<<endl;
1129                    }
1130                    //cout<<"x,y="<<x<<","<<y<<" ix,iy="<<ix<<","<<iy<<" wx,wy="<<wx<<","<<
                            wy<<endl;
1131            }
1132            for(ix=0;ix<=XBINS;++ix){
1133                    hist[ix][0]*=2.0;
1134                    hist[ix][YBINS]*=2.0;
1135            }
1136            for(iy=0;iy<=YBINS;++iy){
1137                    hist[0][iy]*=2.0;
1138                    hist[XBINS][iy]*=2.0;
1139            }
1140
1141    #endif
1142            if(30*errorcount>Ns) cout<<"ERROR many strings outside histogram boundaries"
1143                            <<" errorcount="<<errorcount<<" Ns="<<Ns<<endl;
1144
1145
1146
1147            //Write to data file
1148            stringstream strtime;
1149            strtime<<timestep;
1150
1151            ofstream uthist;
1152            ofstream histm;
1153
1154            if(timestep >= 0 && timestep <= 9) uthist.open(("data/hist/histogram0000"+
                    strtime.str()+".dat").c_str(),ios::trunc);
1155            else if(timestep > 9 && timestep <= 99) uthist.open(("data/hist/histogram000"+
                    strtime.str()+".dat").c_str(),ios::trunc);
1156            else if(timestep > 99 && timestep <= 999) uthist.open(("data/hist/histogram00"+
                    strtime.str()+".dat").c_str(),ios::trunc);
1157            else if(timestep > 999 && timestep <= 9999) uthist.open(("data/hist/histogram0"+
                    strtime.str()+".dat").c_str(),ios::trunc);
1158            else if(timestep > 9999) uthist.open(("data/hist/histogram"+strtime.str()+".dat"
                    ).c_str(),ios::trunc);
1159            else cout << "histogram: timestep not in valid ransge" << timestep << endl;
1160
1161            histm.open("data/hist/histlist.m", ios::app);
1162            histm<<"load("<<'histogram"<<strtime.str()<<".dat')"<<endl;
1163            for(ix=0,x=xl ; ix<=XBINS ; ++ix,x+=xbin){
1164                    for(iy=0,y=yl ; iy<=YBINS ; ++iy,y+=ybin){
1165                            uthist<<hist[ix][iy]/dxdy<<" ";
1166                    }
1167                    uthist<<endl;
1168            }
1169            uthist.close();
1170            histm.close();
1171    }
1172
1173    //Initializes the rawdata.dat file
1174    void initdata(){
1175            ofstream rawdata;
1176            rawdata.open("data/rawdata.dat", ios::trunc);
1177            rawdata.close();
1178    }
1179
1180    //Dumps the raw data to file at calling
1181    void rawdata(double time, int step){
1182            double x=0.0,y=0.0;
1183            ofstream rawdata;
1184            rawdata.open("data/rawdata.dat", ios::app);
1185            rawdata<<time<<" "<<step;
1186            for(int i=0; i<Ns; ++i){
1187                    s[i].getpos(&x,&y);
1188                    rawdata<<" "<<x<<" "<<y;
1189            }
1190            for(int i=Ns; i<Nsmax; ++i){
1191                    rawdata<<" "<<0.0<<" "<<0.0;
1192            }
1193            rawdata<<endl;
1194            rawdata.close();
```

```
1195 }
1196 #ifdef TEST
1197 void initdatatest(){
1198         ofstream rawdata;
1199         rawdata.open("data/rawdata_test.dat", ios::trunc);
1200         rawdata.close();
1201 }
1202
1203 //Dumps the raw data to file at calling
1204 void rawdatatest(double time, int step){
1205         double x=0.0,y=0.0;
1206         ofstream rawdata;
1207         rawdata.open("data/rawdata_test.dat", ios::app);
1208         rawdata<<time<<" "<<step;
1209         for(int i=0; i<Nt; ++i){
1210                 test[i].getpos(&x,&y);
1211                 rawdata<<" "<<x<<" "<<y;
1212                 //cout<<"x="<<x<<" y="<<y<<endl;
1213         }
1214         for(int i=Ns; i<Nsmax; ++i){
1215                 rawdata<<" "<<0.0<<" "<<0.0;
1216         }
1217         rawdata<<endl;
1218         rawdata.close();
1219 }
1220
1221 void initdatav(){
1222         ofstream rawdata;
1223         rawdata.open("data/rawdata_test_v.dat", ios::trunc);
1224         rawdata.close();
1225 }
1226
1227 void rawdatav(double time, int step){
1228         double vx=0.0,vy=0.0;
1229         ofstream rawdata;
1230         rawdata.open("data/rawdata_test_v.dat", ios::app);
1231         rawdata<<setw(12)<<setprecision(10)<<time<<" "<<setw(12)<<step;
1232         for(int i=0; i<Nt; ++i){
1233                 test[i].getv(&vx,&vy);
1234                 rawdata<<" "<<setw(12)<<setprecision(10)<<vx<<" "<<setw(12)<<
                        setprecision(10)<<vy;
1235                 //cout<<"x="<<x<<" y="<<y<<endl;
1236         }
1237         for(int i=Ns; i<Nsmax; ++i){
1238                 rawdata<<" "<<0.0<<" "<<0.0;
1239         }
1240         rawdata<<endl;
1241         rawdata.close();
1242 }
1243
1244 // ExB(double x, double y, double &ExBx, double &ExBy)
1245
1246 void initdata_euler(){
1247         ofstream rawdata;
1248         rawdata.open("data/rawdata_euler.dat", ios::trunc);
1249         rawdata.close();
1250 }
1251
1252 void rawdata_euler(double time, int step, bool eulerfirst){
1253         ofstream rawdata;
1254         rawdata.open("data/rawdata_euler.dat", ios::app);
1255
1256
1257         double vx = 0.0, vy = 0.0;
1258         double xtemp = 0.0, ytemp = 0.0;
1259
1260         if(eulerfirst){
1261                 Xeuler = new double [Nt];
1262                 Yeuler = new double [Nt];
1263                 for(i=0;i<Nt;++i){
1264                         Xeuler[i] = 2*distx*drand48()-distx;
1265                         Yeuler[i] = 2*disty*drand48()-disty;
1266 //                      cout<<"Xtemp="<<Xtemp[i]<<" Ytemp="<<Ytemp[i]<<endl;
1267                 }
1268         }
1269         rawdata<<setw(12)<<setprecision(10)<<time<<" "<<setw(12)<<step;
1270         for(i=0;i<Nt;++i){
1271                 xtemp = Xeuler[i]; ytemp = Yeuler[i];
1272 //              cout<<"xtemp="<<xtemp<<" ytemp="<<ytemp<<endl;
1273                 ExB(xtemp, ytemp, vx, vy);
1274                 rawdata<<" "<<setw(12)<<setprecision(10)<<vx<<" "<<setw(12)<<
                        setprecision(10)<<vy;
1275                 vx = 0.0; vy = 0.0;
```

```
1276                    xtemp = 0.0; ytemp = 0.0;
1277            }
1278            rawdata<<endl;
1279            rawdata.close();
1280
1281
1282    }
1283    #endif
1284
1285    void initdata_hamilton(){
1286            ofstream rawdata;
1287            rawdata.open("data/hamiltonian.dat", ios::trunc);
1288            rawdata.close();
1289    }
1290
1291    void hamiltondata(double time, int step){
1292            ofstream rawdata;
1293            rawdata.open("data/hamiltonian.dat", ios::app);
1294
1295            double hamilton = 0.0;
1296
1297            rawdata<<setw(12)<<setprecision(10)<<time<<" "<<setw(12)<<step;
1298
1299            hamiltonian(hamilton);
1300
1301            rawdata<<" "<<setw(12)<<setprecision(10)<<hamilton<<endl;
1302            rawdata.close();
1303    }
1304
1305
1306    //Closes the hist and mesh files
1307    void fileend(){
1308            ofstream histm("data/hist/histlist.m", ios::app);
1309            ofstream meshE("data/mesh/meshElist.m", ios::app);
1310            ofstream meshEB("data/mesh/meshEBlist.m", ios::app);
1311
1312            histm<<"end"<<endl;
1313            meshE<<"end"<<endl;
1314            meshEB<<"end"<<endl;
1315
1316            histm.close();
1317            meshE.close();
1318            meshEB.close();
1319    }
1320
1321    void parameters(int Nsgen, int Npgen, double Tmax, int n, int Mmax, int Nmax){
1322            //int Mmax, Nmax;
1323            ofstream parameters;
1324            parameters.open("data/parameters.dat", ios::trunc);
1325    # ifdef PERIODIC
1326            parameters<<"Periodic boundary conditions"<<endl;
1327            parameters<<"x_min="<<x_min<<" x_max="<<x_max<<" y_min="<<y_min<<" y_max="<<
                    y_max<<endl;
1328            parameters<<"Mmax="<<Mmax<<" Nmax="<<Nmax<<endl;
1329    #endif
1330    #ifdef BOUNDARY
1331            parameters<<"Destroying boundary conditions";
1332    #endif
1333            parameters<<"Nsmax="<<Nsmax<<" Nsgen="<<Nsgen<<endl;
1334            parameters<<"Npmax="<<Npmax<<" Npgen="<<Npgen<<endl;
1335            parameters<<"Tmax="<<Tmax<<" n="<<n<<endl;
1336            parameters<<"Initial Distribution"<<endl;
1337            parameters<<"x="<<distx<<"drand48()-"<<distx/2.0<<endl;
1338            parameters<<"y="<<disty<<"drand48()-"<<disty/2.0<<endl;
1339            parameters<<"gauss="<<gauss<<endl;
1340            parameters<<"2=> Uniform, 1=>Gaussian Q, 0=> discrete Q"<<endl;
1341    #ifdef BESSEL
1342            parameters<<"Bessel interactions"<<endl;
1343    #else
1344            parameters<<"Logarithmic interactions"<<endl;
1345    #endif
1346            parameters<<"Histogram"<<endl;
1347            parameters<<"Lower x bound="<<xl<<" Upper x bound="<<xu<<" Lower y bound="<<yl<<
                    " Upper y bound="<<yu<<endl;
1348            parameters.close();
1349    }
1350
1351    int main(){
1352            //start timer
1353            clock_t start, finish;
1354            start=clock();
1355            int Nsgen,Npgen,Ntgen;
1356
```

```
1357 |
1358 |#ifdef BOUNDARY
1359 |          Npmax=2;
1360 |          Nsmax=9;
1361 |          Nsgen=Nsmax/2;
1362 |          Npgen=Npmax/2;
1363 |#else
1364 |          Npmax=1;
1365 |          Nsmax=400;
1366 |          Nsgen=Nsmax;
1367 |          Npgen=Npmax;
1368 |#ifdef TEST
1369 |
1370 |          Ntmax=Nsmax/10;
1371 |          Ntgen=Ntmax;
1372 |#else
1373 |          Ntgen = 0;
1374 |#endif
1375 |#endif
1376 |
1377 |
1378 |          double T, t;
1379 |          int n, timestep;
1380 |          declare(T, dt, n);
1381 |          initialize(T,dt);
1382 |          generate(Npgen,Nsgen,Ntgen,true);
1383 |          meshclear();
1384 |          meshforce(0);
1385 |          initmiddel();
1386 |          middel(0.0, 0);
1387 |          initdata();
1388 |          rawdata(0.0, 0);
1389 |          inithist();
1390 |          histogram(0);
1391 |          parameters(Nsgen, Npgen, T, n, Mmax, Nmax);
1392 |#ifdef PERIODIC
1393 |          vmean(0,Mmax);
1394 |#endif
1395 |#ifdef TEST
1396 |          initdatatest();
1397 |          initdatav();
1398 |          rawdatatest(0.0,0);
1399 |          rawdatav(0.0,0);
1400 |          initdata_euler();
1401 |          rawdata_euler(0.0, 0, true);
1402 |          initdata_hamilton();
1403 |          hamiltondata(0.0,0);
1404 |#endif
1405 |          t = 0.0;
1406 |          timestep = 0;
1407 |          while(t<T){
1408 |//        for(t=dt, timestep=1; t<T ; t+=dt, timestep++){
1409 |
1410 |                    movestrings(Mmax, Nmax,t);
1411 |                    moveparticles(Mmax, Nmax);
1412 |//                  setiterationlength();
1413 |                    integrate();
1414 |                    timestep++;
1415 |                    t+=dt;
1416 |
1417 |#ifdef PERIODIC
1418 |                    if(timestep == n/2)corner(Mmax);
1419 |#endif
1420 |#ifdef TEST
1421 |                    if(timestep%10==0){
1422 |                              rawdatatest(t,timestep);
1423 |                              rawdatav(t,timestep);
1424 |                              rawdata_euler(t, timestep, false);
1425 |//                            cout<<"time"<<t<<endl;
1426 |                    }
1427 |                    if(timestep%100==0){
1428 |                              hamiltondata(t,timestep);
1429 |                    }
1430 |
1431 |#endif
1432 |                    if(timestep%10 == 0){
1433 |                              middel(t, timestep);
1434 |                              histogram(timestep);
1435 |                              meshforce(timestep);
1436 |                              rawdata(t, timestep);
1437 |                              cout<<"time="<<t<<endl;
1438 |                    }
1439 |                    printtest = false;
```

```
1440            }
1441            hamiltondata(t,timestep);
1442    #ifdef PERIODIC
1443            vmean(timestep,Mmax);
1444    #endif
1445            //fileend();
1446
1447            delete [] p;
1448            delete [] s;
1449
1450            //stop timer
1451            finish=clock();
1452            cout << "tid = " << ( (finish-start)/(double)CLOCKS_PER_SEC ) << endl;
1453
1454            return 0;
1455    }
```

Listing A.2: StringsandParticles.h,

```
#ifndef STRINGSPARTICLES_H
#define STRINGSPARTICLES_H
//#define BESSEL

#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <sstream>
#include <iomanip>

#include "ODE.cpp"
#include "tbessk.cpp"

//#define PSPRINT

using namespace std;




class Particles{
private:
        int index;
        int N;
        double q,m;
        double *y, *a;
        ODE *ode;
        ofstream output;
public:
        Particles();
        ~Particles();
        void initialize(string header,int in, double X, double Y, double U, double V,
                double Q, double M, double dt,
                void (*derivs)(double*, double*, double*));
        void integrate();
        void print();
        void getE(double X, double Y, double *Ex, double *Ey, int mbox, int nbox, double
             L);
        void getposvel(double *X, double *Y, double *U, double *V);
        void setFq(double Fqx, double Fqy);
        void setall(double X, double Y, double U, double V,double Q,double M);
        void getall(double *X, double *Y, double *U, double *V, double *Q, double *M);
        void setpos(double X, double Y);
        void setdt(double dt);

};

Particles::Particles(){}
Particles::~Particles(){
        output.close();
        delete ode;
        delete [] y;
        delete [] a;
}



class Strings{
private:
        int index;
        int N;
        double *y,*v;
        double B0, Q;
        ODE *ode;
        ofstream output;
public:
        Strings();
        ~Strings();
        void initialize(string header,int in, double X, double Y,
                        double q, double b, double dt,
                        void (*derivs)(double*, double*, double*) );
        void setv(double vx, double vy);
        void getv(double *vx, double *vy);
        void getpos(double *X, double *Y);
        void getEB(double X, double Y, double *Etx, double *Ety, double *Bt, int mbox,
             int nbox, double L,double *distance);
        void integrate();
        void print();
        void setall(double X, double Y, double q, double b);
        void getpotential(double X, double Y, double *pottmp, int mbox, int nbox, double
              L,double Q_temp);
```

```
80          void getall(double *X, double *Y, double *q, double *b);
81          void setpos(double X, double Y);
82          void setdt(double dt);
83  };
84
85
86  Strings::Strings(){}
87  Strings::~Strings(){
88          output.close();
89          delete ode;
90          delete [] y;
91          delete [] v;
92  }
93
94
95
96  #endif //STRINGSPARTICLES_H
```

Listing A.3: Stringsandparticles.cpp,

```
#include "StringsandParticles.h"


/*       *************
 *       *************
 *         Particles
 *       *************
 *       *************
 */

void Particles::initialize(string header, int in, double X, double Y, double U, double V
    ,
                           double Q, double M, double dt,
                           void (*derivs)(double*, double*, double*)){

  /*
  This method is very sensitive to the way the values of y are sent.
  If they are sent as an array, they will only be saved by reference,
  not by value.
  */

  index = in;
  N = 4;
  y = new double[4];
  a = new double[2];

  stringstream fn;
  fn<<"data/particle_"<<index<<".dat";
  output.open(fn.str().c_str(),ios::trunc);



  stringstream cmd;
  cmd<<"p(:,"<<4*index+1<<":"<<4*index+4<<")=load('"<<fn.str()<<"');"<<endl;
  ofstream mfile(header.c_str(),ios::app);
  mfile<<cmd.str();
  mfile.close();


  y[0]=X;
  y[1]=Y;
  y[2]=U;
  y[3]=V;
  a[0]=0.0;
  a[1]=0.0;
  q=Q;
  m=M;
  ode = new ODE(derivs, dt, N);
#ifdef PSPRINT
  cout<<"args="<<X<<" "<<Y<<" "<<U<<" "<<V<<endl;
  cout<<"y="<<y[0]<<" "<<y[1]<<" "<<y[2]<<" "<<y[3]<<endl;
#endif
}


void Particles::integrate(){
  (*ode).rk4(y,a);
#ifdef PSPRINT
  for(int i=0; i<N; ++i) output<<y[i]<<" ";output<<endl;
#endif
}

void Particles::print(){
#ifdef PSPRINT
  cout<<"y="<<y[0]<<" "<<y[1]<<" "<<y[2]<<" "<<y[3]<<endl;
#endif
}

void Particles::getposvel(double *X, double *Y, double *U, double *V){
  //cout<< "Particles::getposvel: i=" <<index<<" X="<<y[0]<<" Y="<<y[1]<<" U="<<y[2]<<"
      V="<<y[3]<<endl;
  *X=y[0];
  *Y=y[1];
  *U=y[2];
  *V=y[3];
}

void Particles::getE(double X, double Y, double *Etx, double *Ety,int mbox, int nbox,
    double L){
  double r=sqrt(pow(X-(y[0]+mbox*L),2)+pow(Y-(y[1]+nbox*L),2));
  //cout<<"Particles::getE r="<<r<<" ("<<X<<","<<Y<<")-("<<y[0]<<","<<y[1]<<")"<<endl;
  //double theta=atan((Y-y[1])/(X-y[0]));
```

```
80    if(r>0){
81      *Etx = q*(X-(y[0]+mbox*L))/r/r;
82      *Ety = q*(Y-(y[1]+nbox*L))/r/r;
83    }else if(r==0.0){
84      *Etx=0.0;
85      *Ety=0.0;
86    }else if(isnan(r)){
87      cout << "ERROR Particles: r is nan: r="<<r<<" EXITING PROGRAM"<<endl;
88      cout <<"X="<<X<<" y[0]="<<y[0]<<" Y="<<Y<<" y[1]="<<y[1]<<endl;
89      exit(1);
90    }else{
91      cout << "ERROR Particles: r<0: r="<<r<<" EXITING PROGRAM"<<endl;
92      cout <<"X="<<X<<" y[0]="<<y[0]<<" Y="<<Y<<" y[1]="<<y[1]<<endl;
93      exit(1);
94    }
95  }
96
97  void Particles::setFq(double Fqx, double Fqy){
98    a[0]=Fqx*q/m;
99    a[1]=Fqy*q/m;
100   //cout<<"Particles::setFq: i="<<index<<" a="<<a[0]<<endl;
101 }
102
103 void Particles::setall(double X, double Y, double U, double V, double Q, double M){
104   y[0] = X;
105   y[1] = Y;
106   y[2] = U;
107   y[3] = V;
108   q = Q;
109   m = M;
110 }
111
112 void Particles::getall(double *X, double *Y, double *U, double *V, double *Q, double *M)
        {
113   *X=y[0];
114   *Y=y[1];
115   *U=y[2];
116   *V=y[3];
117   *Q=q;
118   *M=m;
119 }
120
121 void Particles::setpos(double X, double Y){
122 y[0] = X;
123 y[1] = Y;
124 }
125
126 void Particles::setdt(double dt){
127  (*ode).setdt(dt);
128 }
129
130
131 /*       *************
132  *       *************
133  *         Strings
134  *       *************
135  *       *************
136  */
137
138
139
140
141
142
143
144
145
146 void Strings::initialize(string header, int in, double X, double Y,
147                          double q, double b, double dt,
148                          void (*derivs)(double*, double*, double*)){
149   /*
150   This method is very sensitive to the way the values of y are sent.
151   If they are sent as an array, they will only be saved by reference,
152   not by value.
153   */
154   index =in;
155   N=2;
156   y = new double[2];
157   v = new double[2];
158
159 #ifdef PSPRINT
160   stringstream fn;
161   fn<<"data/string_"<<index<<".dat";
```

```
162     output.open(fn.str().c_str(),ios::trunc);
163
164
165     stringstream cmd;
166     cmd<<"s(:,"<<2*index+1<<":"<<2*index+2<<")=load('"<<fn.str()<<"');"<<endl;
167     ofstream mfile(header.c_str(),ios::app);
168     mfile<<cmd.str();
169     mfile.close();
170 #endif
171
172     y[0]=X;
173     y[1]=Y;
174     v[0]=0.0;
175     v[1]=0.0;
176     Q=q;
177     //cout<<"Strings::initialize Q="<<Q<<endl;
178     B0=b;
179     ode = new ODE(derivs, dt, N);
180 #ifdef PSPRINT
181     cout<<"args="<<X<<" "<<Y<<endl;
182     cout<<"y="<<y[0]<<" "<<y[1]<<endl;
183 #endif
184 }
185 void Strings::setv(double vx, double vy){
186     //cout<<"Strings::setv: i="<<index<<" "<<v[0]<<" "<<v[1]<<endl;
187     v[0]=vx, v[1]=vy;
188 }
189 void Strings::getv(double *vx, double *vy){
190     *vx=v[0]; *vy=v[1];
191 }
192 void Strings::integrate(){
193     //cout<<"Strings::integrate: x="<<y[0]<<" y="<<y[1]<<" u="<<v[0]<<" v="<<v[1]<<endl;
194     (*ode).rk4(y,v);
195     //cout<<"Strings::integrate: x="<<y[0]<<" y="<<y[1]<<" u="<<v[0]<<" v="<<v[1]<<endl;
196 #ifdef PSPRINT
197     for(int i=0; i<N; ++i) output<<y[i]<<" ";output<<endl;
198 #endif
199 }
200 void Strings::print(){
201 #ifdef PSPRINT
202     cout<<"y="<<y[0]<<" "<<y[1]<<endl;
203 #endif
204 }
205
206 void Strings::getpos(double *X, double *Y){
207     *X=y[0];
208     *Y=y[1];
209 }
210
211 void Strings::getEB(double X, double Y, double *Etx, double *Ety, double *Bt, int mbox,
        int nbox, double L, double *distance){
212     double r=sqrt(pow(X-(y[0]+mbox*L),2)+pow(Y-(y[1]+nbox*L),2));
213     *distance = r;
214
215     //cout<<"Strings::getEB r="<<r<<" ("<<X<<","<<Y<<")-("<<y[0]<<","<<y[1]<<")"<<endl;
216     //double theta=atan((Y-y[1])/(X-y[0]));
217 #ifdef BESSEL
218         if(r>0){
219                 *Etx = Q*(X-(y[0]+mbox*L))*BESSK1(r)/r;
220                 *Ety = Q*(Y-(y[1]+nbox*L))*BESSK1(r)/r;
221                 //cout <<"BESSK1(r)="<< BESSK1(r)<<endl;
222                 *Bt = B0;
223         }else if(r==0.0){
224                 *Etx = 0.0;
225                 *Ety = 0.0;
226                 *Bt = B0;
227         }
228 #else
229     if(r>0){
230       *Etx = Q*(X-(y[0]+mbox*L))/r/r;
231       *Ety = Q*(Y-(y[1]+nbox*L))/r/r;
232       *Bt = B0; //TODO
233     }else if(r==0.0){
234       //TODO Special care if r=0!
235       *Etx=0.0;
236       *Ety=0.0;
237       *Bt=B0;
238     }else if(isnan(r)){
239       cout << "ERROR_Strings:_r_is_nan:_r="<<r<<"_EXITING_PROGRAM"<<endl;
240       cout <<"X="<<X<<"_y[0]="<<y[0]<<"_Y="<<Y<<"_y[1]="<<y[1]<<endl;
241       exit(1);
242     }else{
243       cout << "ERROR_Strings:_r<0:_r="<<r<<"_EXITING_PROGRAM"<<endl;
```

```
244     cout <<"X="<<X<<" y[0]="<<y[0]<<" Y="<<Y<<" y[1]="<<y[1]<<endl;
245     exit(1);
246   }
247 #endif
248 }
249
250 void Strings::setall(double X, double Y, double q, double b){
251   y[0] = X;
252   y[1] = Y;
253   Q = q;
254   B0 = b;
255 }
256
257 void Strings::getpotential(double X, double Y, double *pottmp, int mbox, int nbox,
      double L,double Q_temp){
258         double r=sqrt(pow(X-(y[0]+mbox*L),2)+pow(Y-(y[1]+nbox*L),2));
259
260         if(r>0.0){
261
262                 *pottmp = Q_temp*Q*log(r);
263 //              if(sqrt((X-(y[0]+mbox*L))*(X-(y[0]+mbox*L))+(Y-(y[1]+mbox*L))*(Y-(y[1]+
      mbox*L)))==0.0){
264 //                      *pottmp = 0.0;
265 //              }else{
266 //                      *pottmp = -(Q*Q*log(sqrt((X-(y[0]+mbox*L))*(X-(y[0]+mbox*L))+(Y
      -(y[1]+mbox*L))*(Y-(y[1]+mbox*L)))));
267 //              }
268 //              cout<<"Strings::getpotential"<<sqrt((X-(y[0]+mbox*L))*(X-(y[0]+mbox*L))
      +(Y-(y[1]+mbox*L))*(Y-(y[1]+mbox*L)))<<endl;
269 //              cout<<"Strings::getpotential"<<log(sqrt((X-(y[0]+mbox*L))*(X-(y[0]+mbox*
      L))+(Y-(y[1]+mbox*L))*(Y-(y[1]+mbox*L))))<<endl;
270         }else if(r==0.0){
271                 *pottmp = 0.0;
272         }else if(isnan(r)){
273                 cout << "ERROR: Strings::r is nan: r="<<r<<" EXITING PROGRAM"<<endl;
274                 cout <<"X="<<X<<" y[0]="<<y[0]<<" Y="<<Y<<" y[1]="<<y[1]<<endl;
275                 exit(1);
276         }else{
277                 cout << "ERROR: Strings::r<0: r="<<r<<" EXITING PROGRAM"<<endl;
278                 cout <<"X="<<X<<" y[0]="<<y[0]<<" Y="<<Y<<" y[1]="<<y[1]<<endl;
279                 exit(1);
280         }
281 }
282
283 void Strings::getall(double *X, double *Y, double *q, double *b){
284   *X=y[0];
285   *Y=y[1];
286   *q=Q;
287   *b=B0;
288 }
289
290 void Strings::setpos(double X, double Y){
291 y[0] = X;
292 y[1] = Y;
293 }
294
295 void Strings::setdt(double dt){
296  (*ode).setdt(dt);
297  //cout<<"dt3="<<dt<<endl;
298 }
```

Listing A.4: ODE.cpp,

```
#include <iostream>
#include <fstream>
#include <iomanip>

//#define ODE_MAIN

using namespace std;


class ODE{
private:
        int N;
        double h, h2, h6;
        double *yt, *dyt, *dys, *dym;
        void (*derivs)(double *, double *, double *);

public:
        ODE(void (*der)(double *,double *,double *), double d, int nn);
        ~ODE();
        void rk4(double *y, double *F);
        void euler(double *y, double *F);
        void setdt(double d);
};




ODE::ODE(void (*der)(double *,double *,double *), double d, int nn){
        N=nn;
        h=d;
        h2=d/2.0;
        h6=d/6.0;
        yt = new double[N];
        dyt = new double[N];
        dys = new double[N];
        dym = new double[N];
        for(int i=0; i<N; ++i) dys[i]=dyt[i]=dym[i]=0.0;
        derivs = der;

#ifdef DIAG
   cout<<"h2="<<h2<<" h6="<<h6<<endl;
#endif

}

ODE::~ODE(){
        delete [] yt;
        delete [] dyt;
        delete [] dys;
        delete [] dym;
}



void ODE::rk4(double *y, double *F){
        //Zero out dydt-arrays
        int i;
        //Solve with Runge Kuttas 4th order method
        (*derivs)( y, dys, F); //k1
        for(i=0; i<N; i++) yt[i] = y[i]+h2*dys[i];
        (*derivs)( yt, dyt, F); // k2
        for(i=0; i<N; i++) yt[i] = y[i]+h2*dyt[i];
        (*derivs)( yt, dym, F); // k3
        for(i=0; i<N; i++) yt[i] = y[i]+h*dym[i], dym[i] += dyt[i];
        (*derivs)( yt, dyt, F); //k4
        for(i=0; i<N; i++) y[i] += h6*(dys[i]+dyt[i]+2.0*dym[i]);
}


void ODE::euler(double *y, double *F){
        (*derivs)(y, dyt, F);
        for(int i=0; i<N; i++) y[i] += h*dyt[i];
}


void ODE::setdt(double d){
        h=d;
        h2=d/2.0;
        h6=d/6.0;
        //cout<<"dt4="<<h<<endl;
}
```

```
83
84
85
86  #ifdef ODE_MAIN
87  /*
88   *       **** main / testing section ***
89   */
90
91
92  #define XDIM 2
93
94  ofstream output;
95
96  void derivatives(double *y, double *dydt, double *a){
97          dydt[0] = y[1];
98          dydt[2] = y[3];
99          dydt[1] = a[0];
100         dydt[3] = a[1];
101 }
102
103 void print(double *y,int N){
104         for(int i=0; i<N; ++i) output<<setw(12)<<y[i]<<"        "; output<<endl;
105 }
106
107
108 int main(){
109         int N=4;
110         double *y = new double[N];
111         double *a = new double[XDIM];
112         double m=1.0;
113         double dt=0.1;
114
115         output.open("derivatives.dat",ios::trunc);
116         output.precision(4);
117
118         y[0]=0.0; //x0
119         y[1]=1.0; //vx0
120         y[2]=0.0; //y0
121         y[3]=0.0; //vy0
122
123         a[0]=0.0/m;//Fx/m
124         a[1]=1.0/m;//Fy/m
125
126
127         ODE ode(derivatives, dt, N);
128         print(y,N);
129         for(int i=0; i<100; ++i){
130                 ode.rk4(y,a);
131         //      ode.euler(y,a);
132                 print(y,N);
133         }
134
135         output.close();
136         delete [] y;
137         delete [] a;
138         return 0;
139 }
140
141 #endif
```

Appendix B

MATLAB data analysis routines

Listing B.1: histmeshplot.m,

```
clear all; close all;

mycol = [
    1.0000    1.0000    1.0000;
         0         0    1.0000;
         0    0.0455    1.0000;
         0    0.0909    1.0000;
         0    0.1364    1.0000;
         0    0.1818    1.0000;
         0    0.2273    1.0000;
         0    0.2727    1.0000;
         0    0.3182    1.0000;
         0    0.3636    1.0000;
         0    0.4091    1.0000;
         0    0.4545    1.0000;
         0    0.5000    1.0000;
         0    0.5455    1.0000;
         0    0.5909    1.0000;
         0    0.6364    1.0000;
         0    0.6818    1.0000;
         0    0.7273    1.0000;
         0    0.7727    1.0000;
         0    0.8182    1.0000;
         0    0.8636    1.0000;
         0    0.9091    1.0000;
         0    0.9545    1.0000;
         0    1.0000    1.0000;
    0.0625    1.0000    0.9375;
    0.1250    1.0000    0.8750;
    0.1875    1.0000    0.8125;
    0.2500    1.0000    0.7500;
    0.3125    1.0000    0.6875;
    0.3750    1.0000    0.6250;
    0.4375    1.0000    0.5625;
    0.5000    1.0000    0.5000;
    0.5625    1.0000    0.4375;
    0.6250    1.0000    0.3750;
    0.6875    1.0000    0.3125;
    0.7500    1.0000    0.2500;
    0.8125    1.0000    0.1875;
    0.8750    1.0000    0.1250;
    0.9375    1.0000    0.0625;
    1.0000    1.0000         0;
    1.0000    0.9375         0;
    1.0000    0.8750         0;
    1.0000    0.8125         0;
    1.0000    0.7500         0;
    1.0000    0.6875         0;
    1.0000    0.6250         0;
    1.0000    0.5625         0;
    1.0000    0.5000         0;
    1.0000    0.4375         0;
    1.0000    0.3750         0;
    1.0000    0.3125         0;
    1.0000    0.2500         0;
    1.0000    0.1875         0;
    1.0000    0.1250         0;
    1.0000    0.0625         0;
    1.0000         0         0;
    0.9375         0         0;
    0.8750         0         0;
    0.8125         0         0;
    0.7500         0         0;
    0.6875         0         0;
    0.6250         0         0;
    0.5625         0         0;
    0.5000         0         0];


%restoredefaultpath;
%path(path,'Bessel_circle_3/data/plots')
boundary = 0;
xl=-50;yl=-50;xu=50;yu=50;

vidObj = VideoWriter('data/plots/hist.avi');
% vidObj = VideoWriter('hist.avi');
vidObj.FrameRate = 10;
vidObj.Quality = 100;
open(vidObj);

load('middel.dat')

```

```
83  histfiles = dir('data/hist/*histogram*.dat');
84  Efilesx = dir('data/mesh/*Efieldx*.dat');
85  Efilesy = dir('data/mesh/*Efieldy*.dat');
86  ExBxfiles = dir('data/mesh/*ExBx*.dat');
87  ExByfiles = dir('data/mesh/*ExBy*.dat');
88
89
90  forcemeshx = load('data/mesh/Xmesh.dat');
91  forcemeshy = load('data/mesh/Ymesh.dat');
92  histmeshx=load('data/hist/histmeshx.dat');
93  histmeshy=load('data/hist/histmeshy.dat');
94
95  % histfiles = dir('oppgave 1/data/hist/histogram*.dat');
96  % Efilesx = dir('oppgave 1/data/mesh/Efieldx*.dat');
97  % Efilesy = dir('oppgave 1/data/mesh/Efieldy*.dat');
98  % ExBxfiles = dir('oppgave 1/data/mesh/ExBx*.dat');
99  % ExByfiles = dir('oppgave 1/data/mesh/ExBy*.dat');
100 %
101 %
102 % forcemeshx = load('oppgave 1/data/mesh/Xmesh.dat');
103 % forcemeshy = load('oppgave 1/data/mesh/Ymesh.dat');
104 % histmeshx=load('oppgave 1/data/hist/histmeshx.dat');
105 % histmeshy=load('oppgave 1/data/hist/histmeshy.dat');
106
107 set(gca,'NextPlot','replacechildren');
108 %F(numel(histfiles)-3) = struct('cdata',[],'colormap',[]);
109
110 f = figure(1);
111
112 % for ind = 1:numel(histfiles)
113 %     histdata = load(histfiles(ind).name);
114 %     histcuth(ind,:) = histdata(floor(length(histdata)/2),:);
115 %     histcutv(:,ind) = histdata(:,floor(length(histdata)/2));
116 %     forcedatax = 0.0001*load(ExBxfiles(ind).name);
117 %     forcedatay = 0.0001*load(ExByfiles(ind).name);
118 %     Edatax = load(Efilesx(ind).name);
119 %     Edatay = load(Efilesy(ind).name);
120 %     %figure
121 %
122 %     %contourf(histmeshx, histmeshy,histdata);
123 %     I = image([xl xu], [yu yl],histdata);
124 %     set(I, 'CDataMapping','scaled');
125 %     colorbar
126 %     tempMAX=max(max(histdata));
127 %     if ind == 1
128 %         MAX=max(max(histdata));
129 %     end
130 % %     if max(max(histdata)) < MAX/2.0
131 % %         MAX=tempMAX;
132 % %     end
133 %     caxis([0,MAX])
134 %     set(gca,'FontSize',16,'LineWidth',2)
135 %     xlabel('$x(\lambda_D)$','Interpreter','LaTeX')
136 %     ylabel('$y(\lambda_D)$','Interpreter','LaTeX')
137 %     hold on
138 %     g = quiver(forcemeshx,forcemeshy,forcedatax,forcedatay,2, 'Color', [0 0.6 0.2]);
139 %     set(g, 'AutoScale', 'off','LineWidth',2)
140 %     %quiver(forcemeshx,forcemeshy, Edatax, Edatay, 'g')
141 %     legend('\bf E x \bf B')
142 %     if boundary==1
143 %         x=cos(0:2*pi/1000:2*pi);
144 %         y=sin(0:2*pi/1000:2*pi);
145 %         plot(1.5*x,1.5*y,'g');
146 %     end
147 %     hold off
148 %     currFrame = getframe(f);
149 %     writeVideo(vidObj, currFrame);
150 %     %F(ind)=getframe;
151 %
152 % end
153
154
155 for ind = 1:round(numel(histfiles)/10):numel(histfiles)
156     i=2;
157     figure(i)
158     histdata = load(histfiles(ind).name);
159     forcedatax = 100*load(ExBxfiles(ind).name);
160     forcedatay = 100*load(ExByfiles(ind).name);
161     Edatax = load(Efilesx(ind).name);
162     Edatay = load(Efilesy(ind).name);
163     I = surf(histmeshx,histmeshy, histdata);
164 %     I = image([min(histmeshx) max(histmeshx)],([min(histmeshy) max(histmeshy)]),
      histdata');
```

```
165     view(2)
166     shading flat
167     set(I, 'CDataMapping','scaled')
168     colormap(mycol);
169     colorbar
170     set(get(get(I,'Annotation'),'LegendInformation'),'IconDisplayStyle','off')
171     set(gca,'FontSize',24,'LineWidth',2)
172 %       if max(max(histdata)) > 50
173 %           caxis([0,100])
174 %       elseif max(max(histdata)) < 50 && max(max(histdata)) > 20
175 %           caxis([0,50])
176 %       elseif max(max(histdata)) < 20 && max(max(histdata)) > 10
177 %           caxis([0,20])
178 %       else
179 %           caxis([0,6])
180 %       end
181     caxis([0 3]);
182     xlabel('$x(\lambda_D)$','Interpreter','LaTeX')
183     ylabel('$y(\lambda_D)$','Interpreter','LaTeX')
184     hold on
185     g = quiver(forcemeshx,forcemeshy,forcedatax,forcedatay,2, 'Color',[0 0 0]);
186     set(g, 'AutoScale', 'off','LineWidth',2)
187     %quiver(forcemeshx,forcemeshy, Edatax, Edatay, 'g')
188     legend('\bf{E}_x_\bf{B}','Location','NorthEast')
189     axis([xl xu yl yu])
190     title(sprintf('time_=_%f',middel(ind, 1)));
191     set(gcf,'Renderer','painters')
192     hold off
193     filename = sprintf('data/plots/histplot%d.eps',ind);
194     saveas(figure(i),filename,'psc2')
195     i = i+1;
196 end
197
198 close(vidObj)
199
200 %movie(F,0)
201
202
203
204 figure
205 h=plot((middel(:,1)),middel(:,3),'-k');
206 grid on
207 set(h,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
208 xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
209 ylabel('$<|r|>(\lambda_D)$','Interpreter','LaTeX')
210 saveas(h, 'data/plots/Er.eps','psc2')
211 figure
212 h2=plot(middel(:,1),(middel(:,4)),'-k');
213 grid on
214 set(h2,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
215 xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
216 ylabel('$<r^2>(\lambda_D)$','Interpreter','LaTeX')
217 saveas(h2, 'data/plots/Er2.eps','psc2')
218 figure
219 h3=plot(middel(:,1),sqrt(middel(:,4)),'-k');
220 grid on
221 set(h3,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
222 xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
223 ylabel('$\sqrt{<r^2>}(\lambda_D)$','Interpreter','LaTeX')
224 saveas(h3, 'data/plots/Er2.eps','psc2')
225
226 figure
227 h=semilogx(middel(:,1),middel(:,3),'-k');
228 grid on
229 set(h,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
230 xlabel('$\log(t)$','Interpreter','LaTeX')
231 ylabel('$<|r|>(\lambda_D)$','Interpreter','LaTeX')
232 saveas(h, 'data/plots/logxEr.eps','psc2')
233 figure
234 h2=semilogx(middel(:,1),middel(:,4),'-k');
235 grid on
236 set(h2,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
237 xlabel('$log(t)$','Interpreter','LaTeX')
238 ylabel('$<r^2>(\lambda_D)$','Interpreter','LaTeX')
239 saveas(h2, 'data/plots/logxEr2.eps','psc2')
240 figure
241 h3=semilogx(middel(:,1),sqrt(middel(:,4)),'-k');
242 grid on
243 set(h3,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
244 xlabel('$log(t)$','Interpreter','LaTeX')
245 ylabel('$\sqrt{<r^2>}(\lambda_D)$','Interpreter','LaTeX')
246 saveas(h3, 'data/plots/logxEr2.eps','psc2')
247
```

```
figure
h=semilogy(middel(:,1),middel(:,3),'-k');
grid on
set(h,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
ylabel('$log(<|r|>)$','Interpreter','LaTeX')
saveas(h, 'data/plots/logyEr.eps','psc2')
figure
h2=semilogy(middel(:,1),middel(:,4),'-k');
grid on
set(h2,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
ylabel('$log(<r^2>)$','Interpreter','LaTeX')
saveas(h2, 'data/plots/logyEr2.eps','psc2')
figure
h3=semilogy(middel(:,1),sqrt(middel(:,4)),'-k');
grid on
set(h3,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$t(\lambda_D/U_{char})$','Interpreter','LaTeX')
ylabel('$log(\sqrt{<r^2>})$','Interpreter','LaTeX')
saveas(h3, 'data/plots/logyEr2.eps','psc2')

figure
h=loglog(middel(:,1),middel(:,3),'-k');
grid on
set(h,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$log(t)$','Interpreter','LaTeX')
ylabel('$log(<|r|>)$','Interpreter','LaTeX')
saveas(h, 'data/plots/logEr.eps','psc2')
figure
h2=loglog(middel(:,1),middel(:,4),'-k');
grid on
set(h2,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$log(t)$','Interpreter','LaTeX')
ylabel('$log(<r^2>)$','Interpreter','LaTeX')
saveas(h2, 'data/plots/logEr2.eps','psc2')
figure
h3=loglog(middel(:,1),sqrt(middel(:,4)),'-k');
grid on
set(h3,'LineWidth',2);set(gca,'FontSize',24,'LineWidth',2)
xlabel('$log(t)$','Interpreter','LaTeX')
ylabel('$log(\sqrt{<r^2>})$','Interpreter','LaTeX')
saveas(h3, 'data/plots/logEr2.eps','psc2')

histcut1 = histcuth(:,25:-1:1);
histcut2 = histcuth(:,27:51);
histcut3 = histcutv(25:-1:1,:);
histcut4 = histcutv(27:51,:);
cut = (histcut1+histcut2+histcut3.'+histcut4.')./4;
x = linspace(0,150,25);
t=linspace(0,100,1000);
figure
 contourf(x,t,cut,50)
caxis([0,1])
colorbar
xlabel('r')
ylabel('t')
```

Listing B.2: histmeshplot.m,

```
% Directory of the files
d = 'data/mesh/';
% Retrieve the name of the files only
names = dir(d);
names = {names(~[names.isdir]).name};
% Calculate the length of each name and the max length
len  = cellfun('length',names);
mLen = max(len);
% Exclude from renaming the files long as the max
idx   = len < mLen;
len   = len(idx);
names = names(idx);

%Core of the script:
% Rename in a LOOP
for n = 1:numel(names)
    oldname = [d names{n}];
    newname = sprintf('%s%0*s',d,mLen, names{n});
    %movefile(oldname, newname)
    A=java.io.File(oldname);
    A.renameTo(java.io.File(newname));
%     dos(['rename "' oldname '" "' newname '"']); % (1)
end

 A = java.io.File(filename);
   A.renameTo(java.io.File(newFilename));
```

Listing B.3: trajectories4.m,

```
clear all; close all;
restoredefaultpath;
path(path,'r=14.14214/data_b1/')
load('rawdata.dat')
i=0.0;
SIZE=size(rawdata);
MAX = floor(SIZE(2)/8)
figure

for j=1:round(SIZE(1)/20):SIZE(1)
    k = 1;
    h=figure(k);
    hold on
for i=2:MAX+2
    f=plot(rawdata(j,i*2-1),rawdata(j,i*2),'xb');
    set(f,'LineWidth',1)
end
hold off
hold on
for i=MAX+2:2*MAX+1
    f=plot(rawdata(j,i*2-1),rawdata(j,i*2),'xr');
    set(f,'LineWidth',1)
end
hold off
hold on
for i=2*MAX+2:3*MAX+1
    f=plot(rawdata(j,i*2-1),rawdata(j,i*2),'xg');
    set(f,'LineWidth',1)
end
hold off
hold on
for i=3*MAX+2:4*MAX+1
    f=plot(rawdata(j,i*2-1),rawdata(j,i*2),'xm');
    set(f,'LineWidth',1)
end
axis([-100 100 -80 80])
set(gca, 'FontSize',16,'LineWidth',2)
title(sprintf('time_=_%f',rawdata(j, 1)))
xlabel('$x(\lambda_D)$','Interpreter','LaTeX')
ylabel('$y(\lambda_D)$','Interpreter','LaTeX')
grid on
hold off
filename = sprintf('r=14.14214/data_b1/plots/plot%d.eps',j);
saveas(figure(k),filename,'psc2')
close all
k=k+1;
end
```

Listing B.4: trajectories4.m,

```
clear all; close all;
restoredefaultpath;

d =20;
b=1.30*d;
xnorm = 10;
ynorm = 0;
a = ynorm;
c = xnorm;

path(path,'r=7.07107/data_b130d');

data = load('rawdata.dat');

SIZE=size(data);
MAX = floor(SIZE(2)/8);

j = SIZE(1);

nw1 = 0; nw2 = 0; nw3 = 0; nw4 = 0;
ne1 = 0; ne2 = 0; ne3 = 0; ne4 = 0;
sw1 = 0; sw2 = 0; sw3 = 0; sw4 = 0;
se1 = 0; se2 = 0; se3 = 0; se4 = 0;
c1 = 0; c2 = 0; c3 = 0; c4 = 0;


for i=2:MAX+2
    if sqrt((data(j,i*2-1)+c)^2+(data(j,i*2)+a)^2)<10
        c1 = c1+1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)>0
        ne1 = ne1+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)>0
        nw1 = nw1 + 1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)<0
        se1 = se1+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)<0
        sw1 = sw1+1;
    end
end



for i=MAX+2:2*MAX+1
    if sqrt((data(j,i*2-1)+c)^2+(data(j,i*2)+a)^2)<10
        c2 = c2+1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)>0
        ne2 = ne2+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)>0
        nw2 = nw2 + 1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)<0
        se2 = se2+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)<0
        sw2 = sw2+1;
    end
end

for i=2*MAX+2:3*MAX+1
    if sqrt((data(j,i*2-1)+c)^2+(data(j,i*2)+a)^2)<10
        c3 = c3+1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)>0
        ne3 = ne3+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)>0
        nw3 = nw3 + 1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)<0
        se3 = se3+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)<0
        sw3 = sw3+1;
    end
end

for i=3*MAX+2:4*MAX+1
    if sqrt((data(j,i*2-1)+c)^2+(data(j,i*2)+a)^2)<10
        c4 = c4+1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)>0
        ne4 = ne4+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)>0
        nw4 = nw4 + 1;
    elseif data(j,i*2-1)+c>0 && (data(j,i*2)+a)<0
        se4 = se4+1;
    elseif data(j,i*2-1)+c<0 && (data(j,i*2)+a)<0
        sw4 = sw4+1;
    end
```

```
83  end
84
85  fileID = fopen('table2.tex','a');
86  fprintf(fileID,'%6.4g_&_\\begin{tabular}{c|c}\n',b)
87  fprintf(fileID,'%d_&_%d\\\\\\hline\n',ne2,ne1)
88  fprintf(fileID,'%d_&_%d_\\end{tabular}&\n',ne3,ne4)
89  fprintf(fileID,'\\begin{tabular}{c|c}\n')
90  fprintf(fileID,'%d_&_%d\\\\\\hline\n',nw2,nw1)
91  fprintf(fileID,'%d_&_%d_\\end{tabular}&\n',nw3,nw4)
92  fprintf(fileID,'\\begin{tabular}{c|c}\n')
93  fprintf(fileID,'%d_&_%d\\\\\\hline\n',sw2,sw1)
94  fprintf(fileID,'%d_&_%d_\\end{tabular}&\n',sw3,sw4)
95  fprintf(fileID,'\\begin{tabular}{c|c}\n')
96  fprintf(fileID,'%d_&_%d\\\\\\hline\n',se2,se1)
97  fprintf(fileID,'%d_&_%d_\\end{tabular}&\n',se3,se4)
98  fprintf(fileID,'\\begin{tabular}{c|c}\n')
99  fprintf(fileID,'%d_&_%d\\\\\\hline\n',c2,c1)
100 fprintf(fileID,'%d_&_%d_\\end{tabular}\\\\\\hline\n',c3,c4)
101 fclose(fileID)
102
103
104 %
105 %  0 & \begin{tabular}{c|c}
106 %        1 & 2\\\hline
107 %       3 & 4
108 %       \end{tabular} &
109 % \begin{tabular}{c|c}
110 %        1 & 2\\\hline
111 %       3 & 4
112 %       \end{tabular}&
113 % \begin{tabular}{c|c}
114 %        1 & 2\\\hline
115 %       3 & 4
116 %       \end{tabular}&
117 % \begin{tabular}{c|c}
118 %        1 & 2\\\hline
119 %       3 & 4
120 %       \end{tabular}&
121 % \begin{tabular}{c|c}
122 %        1 & 2\\\hline
123 %       3 & 4
124 %       \end{tabular}\\\hline
```

Listing B.5: interpolation.m,

```
clear all; close all;
restoredefaultpath;
tic
run = '1Q';
path(path,sprintf('run_%s/data',run))


%Lagrangian auto- and crosscorrelation functions
vdata = load('rawdata_test_v.dat');
t = vdata(:,1);
ti = linspace(0,max(t),floor(length(t)-1));
SIZE=size(vdata);



for i=3:SIZE(2)
    vi(:,i-2) = interp1(t,vdata(:,i),ti,'linear');
end

SIZE =size(vi);

vx = zeros(SIZE(1),SIZE(2)/2);
vy = vx;

vx(:,1:SIZE(2)/2)=vi(:,1:2:SIZE(2));
vy(:,1:SIZE(2)/2)=vi(:,2:2:SIZE(2));

vx1 = vx(:,1);
vy1 = vy(:,1);

n = length(ti);

% vx = vx1;
% vy = vy1;

%Calculate mean square velocities
for k=1:SIZE(2)/2
    meansqx(k) = mean(vx(:,k).^2);
    meansqy(k) = mean(vy(:,k).^2);
    meansqxy(k) = sqrt(meansqx(k).*meansqy(k));
end

meansquarex = mean(meansqx)
meansquarey = mean(meansqy)
meansquarexy = mean(meansqxy)
meansquareV = meansquarex + meansquarey

msqLx = meansquarex;
msqLy = meansquarey;

%Estimate PDFs of the velocity components and calculate gaussian N(0,sqrt(meansqurex))
%x-component:
%PDF estimate
xn = linspace(floor(min(min(vx))),ceil(max(max(vx))),500);
for k=1:SIZE(2)/2
    FvxL(:,k) = ksdensity(vx(:,k),xn);
end
%Gaussian
FvxL_m = sum(FvxL,2)/(SIZE(2)/2);
normaldistxL = pdf('Normal',xn,0,sqrt(meansquarex));

%y-component:
%PDF estimate
yn = linspace(floor(min(min(vy))),ceil(max(max(vy))),500);
for k=1:SIZE(2)/2
    FvyL(:,k) = ksdensity(vy(:,k),yn);
end
%Gaussian
FvyL_m = sum(FvyL,2)/(SIZE(2)/2);
normaldistyL = pdf('Normal',yn,0,sqrt(meansquarey));

%Calculate normalized lagrangian velocity autocorrelation function.
  N=0;
for k=1:SIZE(2)/2
    for i=1:n;
        vdx = 0.0;
        vdy = 0.0;
        vdxy = 0.0;
        vdyx = 0.0;
        for j=1:(n-i+1)
            vdx = vdx + vx(j,k)*vx(j+i-1,k);
            vdy = vdy + vy(j,k)*vy(j+i-1,k);
```

```
83              vdxy = vdxy + vx(j,k)*vy(j+i-1,k);
84              vdyx = vdyx + vy(j,k)*vx(j+i-1,k);
85
86          end
87          fdx(i) = vdx*(1.0/(n-i));
88          fdy(i) = vdy*(1.0/(n-i));
89          fdxy(i) = vdxy*(1.0/(n-i));
90          fdyx(i) = vdyx*(1.0/(n-i));
91          Rxx(i,k) = fdx(i);
92          Ryy(i,k) = fdy(i);
93          Rxy(i,k) = fdxy(i);
94          Ryx(i,k) = fdyx(i);
95          N=N+1;
96      end
97  end
98  Cxx = sum(Rxx,2);
99  Cxx = Cxx./(SIZE(2)/2)/meansquarex;
100 Cyy = sum(Ryy,2);
101 Cyy = Cyy./(SIZE(2)/2)/meansquarey;
102 Cxy = sum(Rxy,2);
103 Cxy = Cxy./(SIZE(2)/2)/meansquarexy;
104 Cyx = sum(Ryx,2);
105 Cyx = Cyx./(SIZE(2)/2)/meansquarexy;
106
107 %Calculate the integral timescale by integrating the correlation function.
108 %Find first value lower than zero in correlation functions
109 for intmax_x = 1:length(ti)
110     if Cxx(intmax_x)<0.0
111         break;
112     end
113     if intmax_x == length(ti)
114         intmax_x = length(ti)-1
115     end
116 end
117 for intmax_y = 1:length(ti)
118     if Cyy(intmax_y)<0.0
119         break;
120     end
121     if intmax_y == length(ti)
122         intmax_y = length(ti)-1
123     end
124 end
125 %Numerical integration
126 Timescale_L_x=trapz(ti(1:intmax_x),Cxx(1:intmax_x))
127 Timescale_L_y=trapz(ti(1:intmax_y),Cyy(1:intmax_y))
128
129
130
131 %Calculating the Lagrangian power spectrum by discrete fourier transform of the
132 %correlation functions
133 NF = floor(n*2/3)
134 Fs = 1/(ti(2)-ti(1));
135 % Sxx = dct(Cxx(1:NFFT),NFFT)/NFFT;
136 % Syy = dct(Cyy(1:NFFT),NFFT)/NFFT;
137 %freq = Fs*linspace(0,1,NFFT);
138
139 fcxx = Cxx(1:NF);
140 fcyy = Cyy(1:NF);
141
142 fcxx = [fcxx(1:length(fcxx)); fcxx(length(fcxx)-1:-1:2)];
143 fcyy = [fcyy(1:length(fcyy)); fcyy(length(fcyy)-1:-1:2)];
144
145 % fcxx = [fcxx(length(fcxx):-1:1); fcxx(2:length(fcxx)-1)];
146 % fcyy = [fcyy(length(fcyy):-1:1); fcyy(2:length(fcyy)-1)];
147
148 NFFT = length(fcxx);
149 freq = Fs*linspace(0,1,NFFT);
150
151 CSxx = fft(fcxx,NFFT)/Fs;
152 CSyy = fft(fcyy,NFFT)/Fs;
153
154 Sxx = real(CSxx);
155 Syy = real(CSyy);
156
157 ISxx = max(imag(CSxx))
158 ISyy = max(imag(CSyy))
159
160 %The timescale is the first element of the power spectrum
161 Timescale_PSD_x = Sxx(1)
162 Timescale_PSD_y = Syy(1)
163
164 %Calculate the effective diffusion coeffisient D=2<v^2>(\tau_lx+\tau_ly)*0.5
165 D = 2*meansquareV*(Timescale_PSD_x+Timescale_PSD_y)*0.5
```

```
166
167 %Calculate <r^2(t)>=2t<v^2>\int_0^t(1-\tau/t)R_L(\tau)d\tau
168 for j=1:n
169     rsq(j) =2*meansquareV*sum((ti(j)-ti(1:j))'.*(Cxx(1:j)+Cyy(1:j))*0.5)*1/(Fs);
170
171 end
172 sqrsq = sqrt(rsq);
173
174 %Plot correlation functions
175 fig1 = figure;
176 set(gcf,'Renderer','painters')
177 h=semilogy(ti,Cxx)
178 set(h,'LineWidth',2)
179 hold on
180 h=semilogy(ti,Cyy,'r')
181 set(h,'LineWidth',2)
182 %h=semilogy(ti,Cxy, 'g')
183 %set(h,'LineWidth',2)
184 %hold on
185 %h=semilogy(ti,Cyx,'c')
186 %set(h,'LineWidth',2)
187 legend('Rxx', 'Ryy','Rxy','Ryx','Location','SouthWest')
188 %legend('Rxx', 'Ryy','Rxy','Ryx','Location','NorthEast')%Legend for 3
189 hold off
190 set(gca,'LineWidth',2,'FontSize',16)
191 grid on
192 if min(Cxx)<0 || min(Cyy)<0
193     axisymin = 1e-3;
194 else
195     axisymin = min([min(Cxx) min(Cyy)]);
196 end
197 axis([0 max(ti) 1e-2 1])
198 title('Normalized_Lagrangian_velocity_autocorrelation_functions')
199 % saveas(fig1,sprintf('run_%s/data/plots/AcorrL.fig',run),'fig')
200 % saveas(fig1,sprintf('run_%s/data/plots/AcorrL.eps',run),'psc2')
201
202 %Plot PDF estimates and fits
203 probdensfuncL = figure;
204 p1 = semilogy(xn,FvxL_m,'b');
205 set(p1,'LineWidth',2)
206 hold on;
207 norm1 = semilogy(xn,normaldistxL,'--b')
208 set(norm1,'LineWidth',2)
209 p2 = semilogy(yn,FvyL_m,'r');
210 set(p2,'LineWidth',2)
211 norm2 = semilogy(yn,normaldistyL,'--r')
212 set(norm2,'LineWidth',2)
213 set(gca,'LineWidth',2,'FontSize',16)
214 pleg = legend('v_x_PDF_estimate','N(0,sqrt(<v_x^2>))','v_y_PDF_estimate','N(0,sqrt(<v_y
      ^2>))')
215 set(pleg,'Interpreter','tex')
216 axis([-3 3 1e-5 max([max(FvxL_m) max(FvyL_m)])])
217 title('Lagrangian_velocity_PDFs')
218 xlabel('$v_(\lambda_D/t_0)$','Interpreter','LaTeX')
219 ylabel('Probability')
220 grid on
221 hold off;
222 % saveas(probdensfuncL,sprintf('run_%s/data/plots/PDFL.fig',run),'fig')
223 % saveas(probdensfuncL,sprintf('run_%s/data/plots/PDFL.eps',run),'psc2')
224
225 %Plot power spectra
226 powerspectra = figure;
227 s1 = loglog(freq(1:floor(length(freq)/2)),smooth(Sxx(1:floor(length(Sxx)/2)),10),'b')
228 set(s1,'LineWidth',2)
229 hold on
230 s2 = loglog(freq(1:floor(length(freq)/2)),smooth(Syy(1:floor(length(Syy)/2)),10),'r')
231 set(s2,'LineWidth',2)
232 sfit = loglog(freq,(freq.^(-2))./9,'--k')
233 set(sfit,'LineWidth',2)
234 set(gca,'LineWidth',2,'FontSize',16)
235 grid on;
236 xlabel('Frequency_(1/t_0)')
237 ylabel('Power')
238 legend('G_{xx}','G_{yy}','Freq^{-2}')
239 title('Lagrangian_power_spectral_density')
240 hold off
241 axis([7e-3 1e2 1e-8 1e2])%axis for 1
242 % saveas(powerspectra,sprintf('run_%s/data/plots/psdL.fig',run),'fig')
243 % saveas(powerspectra,sprintf('run_%s/data/plots/psdL.eps',run),'psc2')
244
245 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
246 %break;
247
```

```
%Eulerian auto- and cross correlation functions
vdata = load('rawdata_euler.dat');
t = vdata(:,1);
ti = linspace(0,max(t),length(t)-1);
SIZE=size(vdata);

for i=3:SIZE(2)
    vi(:,i-2) = interp1(t,vdata(:,i),ti,'linear');
end

SIZE =size(vi);

vx = zeros(SIZE(1),SIZE(2)/2);
vy = vx;

vx(:,1:SIZE(2)/2)=vi(:,1:2:SIZE(2));
vy(:,1:SIZE(2)/2)=vi(:,2:2:SIZE(2));

vx1 = vx(:,1);
vy1 = vy(:,1);

n = length(ti);

% vx = vx1;
% vy = vy1;

for k=1:SIZE(2)/2
    meansqx(k) = mean(vx(:,k).^2);
    meansqy(k) = mean(vy(:,k).^2);
    meansqxy(k) = sqrt(meansqx(k).*meansqy(k));
end

meansquarex = mean(meansqx)
meansquarey = mean(meansqy)
meansquarexy = mean(meansqxy)

msqEx = meansquarex;
msqEy = meansquarey;

xn = linspace(floor(min(min(vx))),ceil(max(max(vx))),100);
for k=1:SIZE(2)/2
    FvxE(:,k) = ksdensity(vx(:,k),xn);
end
FvxE_m = sum(FvxE,2)/(SIZE(2)/2);
normaldistxE = pdf('Normal',xn,0,sqrt(meansquarex));

yn = linspace(floor(min(min(vy))),ceil(max(max(vy))),100);
for k=1:SIZE(2)/2
    FvyE(:,k) = ksdensity(vy(:,k),yn);
end
FvyE_m = sum(FvyE,2)/(SIZE(2)/2);
normaldistyE = pdf('Normal',yn,0,sqrt(meansquarey));

  N=0;
for k=1:SIZE(2)/2
    for i=1:n;
        vdx = 0.0;
        vdy = 0.0;
        vdxy = 0.0;
        vdyx = 0.0;
        for j=1:(n-i+1)
            vdx = vdx + vx(j,k)*vx(j+i-1,k);
            vdy = vdy + vy(j,k)*vy(j+i-1,k);
            vdxy = vdxy + vx(j,k)*vy(j+i-1,k);
            vdyx = vdyx + vy(j,k)*vx(j+i-1,k);

        end
        fdx(i) = vdx*(1.0/(n-i));
        fdy(i) = vdy*(1.0/(n-i));
        fdxy(i) = vdxy*(1.0/(n-i));
        fdyx(i) = vdyx*(1.0/(n-i));
        Rxx(i,k) = fdx(i);
        Ryy(i,k) = fdy(i);
        Rxy(i,k) = fdxy(i);
        Ryx(i,k) = fdyx(i);
        N=N+1;
    end
end
Cxx = sum(Rxx,2);
Cxx = Cxx./(SIZE(2)/2)/meansquarex;
Cyy = sum(Ryy,2);
Cyy = Cyy./(SIZE(2)/2)/meansquarey;
Cxy = sum(Rxy,2);
```

```
Cxy = Cxy./(SIZE(2)/2)/meansquarexy;
Cyx = sum(Ryx,2);
Cyx = Cyx./(SIZE(2)/2)/meansquarexy;

N;

%Calculating the Eulerian power spectrum by discrete fourier transform of the
%correlation functions
NF = floor(n*2/3)
Fs = 1/(ti(2)-ti(1));
% Sxx = dct(Cxx(1:NFFT),NFFT)/NFFT;
% Syy = dct(Cyy(1:NFFT),NFFT)/NFFT;
%freq = Fs*linspace(0,1,NFFT);

fcxx = Cxx(1:NF);
fcyy = Cyy(1:NF);

fcxx = [fcxx(1:length(fcxx)); fcxx(length(fcxx)-1:-1:2)];
fcyy = [fcyy(1:length(fcyy)); fcyy(length(fcyy)-1:-1:2)];

% fcxx = [fcxx(length(fcxx):-1:1); fcxx(2:length(fcxx)-1)];
% fcyy = [fcyy(length(fcyy):-1:1); fcyy(2:length(fcyy)-1)];

NFFT = length(fcxx);
freq = Fs*linspace(0,1,NFFT);

CSxx = fft(fcxx,NFFT)/Fs;
CSyy = fft(fcyy,NFFT)/Fs;

Sxx = real(CSxx);
Syy = real(CSyy);

ISxx = max(imag(CSxx))
ISyy = max(imag(CSyy))

%The timescale is the first element of the power spectrum
Timescale_E_PSD_x = Sxx(1)
Timescale_E_PSD_y = Syy(1)

fig1 = figure;

h=loglog(ti,Cxx,'b')
set(h,'LineWidth',2)
hold on
h=loglog(ti,Cyy,'r')
set(h,'LineWidth',2)
%h=loglog(ti,Cxy, 'g')
set(h,'LineWidth',2)
%h=loglog(ti,Cyx,'c')
set(h,'LineWidth',2)
legend('Rxx', 'Ryy','Cxy','Cyx', 'Location', 'SouthWest')
hold off
set(gca,'LineWidth',2,'FontSize',16)
title('Normalized Eulerian velocity autocorrelation functions')
grid on
if min(Cxx)<0 || min(Cyy)<0
    axisyminE = 1e-3;
elseif min(Cxx)<0.1 || min(Cyy)<0.1
    axisyminE = min([min(Cxx) min(Cyy)]);
else
    axisyminE = 0.1;
end
axis([1e-3 max(ti) 1e-2 max([max(Cxx(1:n-0.1*n)) max(Cyy(1:n-0.1*n))])])
set(gca,'YMinorTick','on')
% saveas(fig1,sprintf('run_%s/data/plots/AcorrE.fig',run),'fig')
% saveas(fig1,sprintf('run_%s/data/plots/AcorrE.eps',run),'psc2')


probdensfuncE = figure;
p1 = semilogy(xn,FvxE_m,'b');
set(p1,'LineWidth',2)
hold on
norm1 = semilogy(xn,normaldistxE,'--b')
set(norm1,'LineWidth',2)
p2 = semilogy(yn,FvyE_m,'r');
set(p2,'LineWidth',2)
norm2 = semilogy(yn,normaldistyE,'--r')
set(norm2,'LineWidth',2)
set(gca,'LineWidth',2,'FontSize',16)
pleg = legend('v_x PDF estimate','N(0,sqrt(<v_x^2>))','v_y PDF estimate','N(0,sqrt(<v_y
    ^2>))')
set(pleg,'Interpreter','tex')
axis([-3 3 1e-5 max([max(FvxE_m) max(FvyE_m)])])
```

```
413 %axis([-7 7 1e-4 1.5])%axis for 05Q
414 title('Eulerian velocity PDFs')
415 xlabel('$v_(\lambda_D/t_0)$','Interpreter','LaTeX')
416 ylabel('Probability')
417 grid on
418 hold off
419 % saveas(probdensfuncE,sprintf('run_%s/data/plots/PDFE.fig',run),'fig')
420 % saveas(probdensfuncE,sprintf('run_%s/data/plots/PDFE.eps',run),'psc2')
421
422 %Plot power spectra
423 powerspectra = figure;
424 s1 = loglog(freq(1:floor(length(freq)/2)),smooth(Sxx(1:floor(length(Sxx)/2)),10),'b')
425 set(s1,'LineWidth',2)
426 hold on
427 s2 = loglog(freq(1:floor(length(freq)/2)),smooth(Syy(1:floor(length(Syy)/2)),10),'r')
428 set(s2,'LineWidth',2)
429 sfit = loglog(freq,(freq.^(-1))./9,'--k')
430 set(sfit,'LineWidth',2)
431 set(gca,'LineWidth',2,'FontSize',16)
432 grid on;
433 xlabel('Frequency (1/t_0)')
434 ylabel('Power')
435 legend('G_{xx}','G_{yy}','Freq^{-1}')
436 title('Eulerian power spectral density')
437 hold off
438 axis([5e-3 1e2 1e-5 1e2])%axis for 1
439 % saveas(powerspectra,sprintf('run_%s/data/plots/psdE.fig',run),'fig')
440 % saveas(powerspectra,sprintf('run_%s/data/plots/psdE.eps',run),'psc2')
441
442
443
444 %%Write mean square velocities L & E to file for comparison. Used on the
445 %%data with random seed
446 fileID = fopen('vmeansquaredata.dat','a');
447 fprintf(fileID,'%d %f %f %f %f\n',((run)+1),msqLx,msqLy,msqEx,msqEy);
448 fclose(fileID);
449
450 %%Write the E&L PSD timescales to file. For use with the random seed data.
451 % fileID=fopen('ELtimescales.dat','a');
452 % fprintf(fileID,'%d %f %f %f %f\n',((run)+1),Timescale_PSD_x,Timescale_PSD_y,
        Timescale_E_PSD_x,Timescale_E_PSD_y);
453 % fclose(fileID);
454
455 %%Write sqrt(<r^2(t)>) to file. Used for the runs with different Q
456 % fileID = fopen(sprintf('rms_r(t)%s.dat',run),'w');
457 % A = [ti; sqrsq];
458 % fprintf(fileID,'%f %f\n',A);
459 % fclose(fileID);
460 %
461 %%Write the mean square velocities (L) to file. Used for the runs with
462 %%different Q
463 % fileID = fopen('vmeansquare_varQ.dat','a');
464 % fprintf(fileID,'%f\n',meansquareV)
465 % fclose(fileID)
466 %
467 %%Write the numerically integrated timescales to file. For the runs with
468 %%different Q
469 % fileID = fopen('timescales_integrated.dat','a');
470 % fprintf(fileID,'%s %f %f\n',run,Timescale_L_x,Timescale_L_y);
471 % fclose(fileID);
472 %
473 %%Write the PSD timescales to file. For the runs with
474 %%different Q
475 % fileID = fopen('timescales_psd(0).dat','a');
476 % fprintf(fileID,'%s %f %f\n',run,Timescale_PSD_x,Timescale_PSD_y);
477 % fclose(fileID);
478 %
479 %%Write the diffusion coeffisients to file. For the runs with different Q
480 % fileID = fopen('diffusion_coeffisients.dat','a');
481 % fprintf(fileID,'%s %f\n',run,D);
482 % fclose(fileID)
483
484
485
486 restoredefaultpath;
487
488 toc
489 restoredefaultpath;
```

Listing B.6: meansquareplot.m,

```
clear all; close all;

vmean = load('vmeansquaredata.dat');
vmeans = [];

for i=1:size(vmean,1)
    vmeans = [vmeans; vmean(i,2) vmean(i,4); vmean(i,3) vmean(i,5)];
end
figure;
hold on;
for i=1:size(vmeans,1)/2
    h = plot(vmeans(i*2-1:i*2,1),vmeans(i*2-1:i*2,2),'-xk');
    set(h,'LineWidth',2)
end
h=plot(vmeans(:,1),vmeans(:,2),'x')
h=plot(vmean(:,2),vmean(:,4),'xb');
set(h,'LineWidth',2)
h=plot(vmean(:,3),vmean(:,5),'xr');
set(h,'LineWidth',2)
set(gca,'LineWidth',2,'FontSize',16)
xlabel('<v_L^2>')
ylabel('<v_E^2>')
axis([0 0.22 0 0.8]);
grid on
h=plot(0.1009,0.2107,'sb')
set(h,'LineWidth',3,'MarkerSize',6)
h=plot(0.0941,0.2088,'sr')
set(h,'LineWidth',3,'MarkerSize',6)
h=plot([0:0.01:0.22],[0:0.01:0.22],'-.k')
set(h,'LineWidth',2)
hold off

% uL = (vmean(:,2)+vmean(:,3))*0.5;
% uE = (vmean(:,4)+vmean(:,5))*0.5;
%
% figure
% h = plot(uL,uE,'x')
```

Listing B.7: timescaleplot.m,

```
clear all; close all;
t = load('ELtimescales.dat');
tml = mean([t(:,2) t(:,3)],2);
tme = mean([t(:,4) t(:,5)],2);
hold on
h=plot(tml,tme,'o');
set(h,'LineWidth',6,'MarkerSize',4)
h =plot(mean(tml),mean(tme),'+r');
set(h,'LineWidth',3,'MarkerSize',10)
set(gca,'LineWidth',2,'FontSize',16)
h=plot(0.5*(13.998705+8.900086),0.5*(14.201664+2.066290e+01),'s')
set(get(get(h,'Annotation'),'LegendInformation'),'IconDisplayStyle','off')
set(h,'LineWidth',6,'MarkerSize',4)
grid on
fito = fit(tml,tme,'poly1');
h=plot(fito,'--k')
set(h,'LineWidth',2)
axis([0 16 0 18])
legend('(\tau_L,\tau_E)','(<\tau_L>,<\tau_E>)',sprintf('y=%6.3gx_+%6.3g',fito.p1,fito.p2
    ),'Location','NorthWest')
xlabel('\tau_L')
ylabel('\tau_E')
hold off
```

Listing B.8: timescales.m,

```
clear all; close all;

T_in = load('timescales_integrated.dat');
T_psd = load('timescales_psd(0).dat');
Dc = load('diffusion_coeffisients.dat');
D = Dc(:,2);
vmsq = load('vmeansquare_varQ.dat');
q = [0.1 0.05 0.025 0.0125 0.00625];



fig1 = figure;
h = loglog(q,T_in(:,2),'-xb')
set(h,'LineWidth',2)
hold on
h = loglog(q,T_in(:,3),'-xr')
set(h,'LineWidth',2)
h = loglog(q,0.5.*(T_in(:,2)+T_in(:,3)),'-xk')
set(h,'LineWidth',2)
h = loglog(q,1./q,'--k')
set(h,'LineWidth',2)
h = loglog(q,2./sqrt(q),'-.k')
set(h,'LineWidth',2)
set(gca,'LineWidth',2,'FontSize',16)
xlabel('Vortex strength (\gamma)')
ylabel('Timescale (\gamma)')
legend('\tau_{Lx}','\tau_{Ly}','\tau_L','\gamma^{-1}','2*\gamma^{-1/2}','Location','
    SouthWest')
title('Numerically integrated timescales')
axis([0.00625 0.1 min(T_in(:,3)) max(T_in(:,2))])
grid on
hold off
saveas(fig1,'timescales.fig','fig')
saveas(fig1,'timescales.eps','psc2')

fig2 = figure;
h = loglog(q,T_psd(:,2),'-xb')
set(h,'LineWidth',2)
hold on
h = loglog(q,T_psd(:,3),'-xr')
set(h,'LineWidth',2)
h = loglog(q,0.5.*(T_psd(:,2)+T_psd(:,3)),'-xk')
set(h,'LineWidth',2)
h = loglog(q,1./q,'--k')
set(h,'LineWidth',2)
h = loglog(q,2./sqrt(q),'-.k')
set(h,'LineWidth',3)
set(gca,'LineWidth',2,'FontSize',16)
xlabel('Vortex strength (\gamma)')
ylabel('Timescale (\gamma)')
legend('\tau_{Lx}','\tau_{Ly}','\tau_L','\gamma^{-1}','2\gamma^{-1/2}','Location','
    SouthWest')
title('Timescales from PSD(f=0)')
axis([0.00625 0.1 min(T_psd(:,3)) max(T_psd(:,2))])
grid on
hold off
saveas(fig2,'timescales_psd.fig','fig')
saveas(fig2,'timescales_psd.eps','psc2')

fig3 = figure;
h = loglog(q,D,'-xb')
set(h,'LineWidth',2)
hold on
h = loglog(q,30*5.5*q.^(3/2),'--k')
set(h,'LineWidth',2)
set(gca,'LineWidth',2,'FontSize',16)
axis([0.00625 0.1 min(D) max(D)])
xlabel('Vortex strength (\gamma)')
ylabel('Diffusion coeffisient')
legend('D(\gamma)','165*\gamma^{3/2}','Location','SouthEast')
grid on
hold off
saveas(fig3,'diffusion_coeffisients_Q.fig','fig')
saveas(fig3,'diffusion_coeffisients_Q.eps','psc2')

fig4 = figure;

h = loglog(q,10*0.5*q,'--k')
set(h,'LineWidth',2)
hold on
h = loglog(q,sqrt(vmsq),'-xb')
set(h,'LineWidth',2)
```

```
81  set(gca,'LineWidth',2,'FontSize',16)
82  xlabel('Vortex strength (\gamma)')
83  ylabel('(<v_L^2>)^{1/2}')
84  legend('5*\gamma','(<v_L^2>)^{1/2}','Location','SouthEast')
85  axis([min(q) max(q) min(sqrt(vmsq)) max(sqrt(vmsq))])
86  grid on
87  hold off
88  saveas(fig4,'rmsv.fig','fig')
89  saveas(fig4,'rmsv.eps','psc2')
90
91  fig5 = figure;
92  h = loglog(sqrt(vmsq),D,'-xb')
93  set(h,'LineWidth',2)
94  hold on
95  h = loglog(sqrt(vmsq),7*(vmsq.^(1/2)),'--k')
96  set(h,'LineWidth',2)
97  %h=loglog()
98  set(gca,'LineWidth',2,'FontSize',16)
99  xlabel('(<v_L^2>)^{1/2}')
100 ylabel('Diffusion coeffisient')
101 legend('D','7*(<v_L^2>)^{1/2}','Location','SouthEast')
102 %axis([1e-2 1e0 1e-1 1e1])
103 axis([min(sqrt(vmsq)) max(sqrt(vmsq)) 1e-1 1e1])
104 grid on
105 hold off
106 saveas(fig5,'diffusion_coeffisients_vrms.fig','fig')
107 saveas(fig5,'diffusion_coeffisients_vrms.eps','psc2')
```

Listing B.9: rmsr.m,

```
clear all; close all;

rms1q = load('rms_r(t)1Q.dat');
rms05q = load('rms_r(t)05Q.dat');
rms025q = load('rms_r(t)025Q.dat');
rms0125q = load('rms_r(t)0125Q.dat');
rms00625q = load('rms_r(t)00625Q.dat');

fig = figure
h=loglog(rms1q(:,1),rms1q(:,2))
set(h,'LineWidth',2)
hold on
h=loglog(rms05q(:,1),rms05q(:,2),'r')
set(h,'LineWidth',2)
h=loglog(rms025q(:,1),rms025q(:,2),'g')
set(h,'LineWidth',2)
h=loglog(rms0125q(:,1),rms0125q(:,2),'c')
set(h,'LineWidth',2)
h=loglog(rms00625q(:,1),rms00625q(:,2),'m')
set(h,'LineWidth',2)
h = loglog(rms1q(:,1),0.5*rms1q(:,1),'--k')
set(h,'LineWidth',2)
h = loglog(rms1q(:,1),1.5*sqrt(rms1q(:,1)),'--k')
set(h,'LineWidth',2)
set(gca,'LineWidth',2,'FontSize',16)
grid on
xlabel('t_(t_0)')
ylabel('(<r^2(t)>)^{1/2}')
axis([2e-2 1e2 1e-2 3e1])
legend('\gamma=0.1','\gamma=0.05','\gamma=0.025','\gamma=0.0125','\gamma=0.00625','0.5*t
    ','1.5*t^{1/2}','Location','NorthWest')
title('Root_mean_square_displacement_for_varying_values_of_Q')
hold off
saveas(fig,'rms_r.eps','psc2');
saveas(fig,'rms_r.fig','fig');
```

Listing B.10: trajectories.m,

```
clear all;close all;

load('rawdata.dat')
%load('rawdata_test.dat')
figure

SIZE=size(rawdata);
%set(gcf,'DefaultAxesColorOrder',[0 0 1; 1 0 0])

    h=plot(rawdata(:,3),rawdata(:,4),'b');
    set(h,'LineWidth',2)
    hold on
    h=plot(rawdata(:,5),rawdata(:,6),'r');
    set(h,'LineWidth',2)
    h=plot(rawdata(1,3),rawdata(1,4),'<b');
    set(h,'LineWidth',4)
    h=plot(rawdata(1,5),rawdata(1,6),'>r');
    set(h,'LineWidth',4)
    h=plot(rawdata(:,7),rawdata(:,8),'b');
    set(h,'LineWidth',2)
    h=plot(rawdata(:,9),rawdata(:,10),'r');
    set(h,'LineWidth',2)
    h=plot(rawdata(1,7),rawdata(1,8),'>b');
    set(h,'LineWidth',4)
    h=plot(rawdata(1,9),rawdata(1,10),'<r');
    set(h,'LineWidth',4)


% SIZET=size(rawdata_test);
% for i=2:SIZET(2)/2
%     h=plot(rawdata_test(:,i*2-1),rawdata_test(:,i*2),'r');
%     set(h,'LineWidth',2)
%     hold all
% end
%     h2=plot(rawdata(length(rawdata),3),rawdata(length(rawdata),4),'x')
%     h3=plot(rawdata(length(rawdata),5),rawdata(length(rawdata),6),'x')
%     h4=plot(rawdata(length(rawdata),7),rawdata(length(rawdata),8),'x')
%     set(h2,'LineWidth',4)
%     set(h3,'LineWidth',4)
%     set(h4,'LineWidth',4)
%     h5=plot(rawdata(1,3),rawdata(1,4),'^');
%     h6=plot(rawdata(1,5),rawdata(1,6),'v');
%     %h7=plot(rawdata(1,7),rawdata(1,8),'o');
%     set(h5,'LineWidth',4)
%     set(h6,'LineWidth',4)
%     %set(h7,'LineWidth',4)
set(gca,'FontSize',16,'LineWidth',2)
xlabel('$x(\lambda_D)$','Interpreter','LaTeX')
ylabel('$y(\lambda_D)$','Interpreter','LaTeX')
%legend('Q=1.0','Q=-0.9')
grid on
axis equal
%axis([-0.5 1.5 -0.1 2.1])
```